It is not immediately obvious that Moore's law should favor one type of processor architecture over another. Why, starting in 1980, did killer micros have such success? Which processor architectures died off? What is a killer micro, anyway? Does _every_ microprocessor count as a killer micro?

The Cray vector architectures died off because they were too expensive, and also because parallel vector processors (i.e., computers) ran into trouble scaling to larger number of processors. But parallel vector processors were supercomputers, pioneering advanced architectural ideas with exotic technologies. More down to earth, DEC's Vax architecture ran into trouble because its complex instruction set made it very hard to implement fast pipelining. (The Vax can stand for other machines that lost their appeal). In 1980, the idea of RISC (reduced instruction set computing) architecture gained ground and led to radical simplifications in the implementation of pipelining.

In the 1970s, machines had hundreds of different instructions, in a variety of formats, leading to thousands of distinct combinations when addressing-mode variations were taken into account. Interpretation of all possible combinations of opcodes and operands required very complex control circuitry. Early VLSI chips simply did not have enough room to hold both this massive control circuitry and a register file with sufficiently many registers. RISC machines were successful because they radically reduced the amount of control circuitry, i.e., they reduced the control _overhead_.

Moreover, the simplicity of control in RISC machines allowed a fully hardwired implementation of the control circuitry, breaking with the tradition of "software" microprogrammed control. This dramatically increased execution speed.

The distinction between RISC and CISC has not stayed hard and fast since 1980, especially after the introduction of Intel chips in the 1990s that were CISC on the outside, and RISC on the inside. Judged by the number of chips sold, we have to count more recent Intel and AMD processors as killer micros. This is so even if they have processor architectures only a mother could love. Another concern is that CISC processors are incredibly expensive and time-consuming to design. You almost have to admire Intel for not being killed by the complexity of its own designs. Whether this is a good starting point for multicore, or, indeed, for any _system on a chip_, is another question entirely. Finally, describing CISC architectures to students is not exactly enjoyable.

We can sketch some of the core ideas in RISC design philosophy, which is both an approach to instruction-set design, and a set of implementation strategies.

1. There shall be a small set of instructions, each of which can be executed in approximately the same amount of time using hardwired control (you may need several RISC instructions to do the work of one complex instruction).

2. The architecture shall be a _load/store_ architecture that confines memory-address calculation, and memory-latency delays, to a small set of load and store instructions, with all other (register-register) instructions obtaining their operands from faster, and compactly addressable, processor registers.

3. There shall be a limited number of simple addressing modes that eliminate or speed up address calculations for the vast majority of cases.

4. There shall be simple, uniform instruction formats that facilitate extraction/decoding of the various fields. This allows overlap between opcode interpretation and register readout.

Admittedly, what ultimately counts are the run times of our programs. What matters is not whether the computer executes 10 billion simple instructions, or 5 billion complex instructions, but simply what the total run time is.

We can characterize RISC architectures in slightly different language (redundancy is not bad):

1. All operations on data apply to data in registers.

2. The only operators that affect memory are loads (which move data from memory to a register) and stores (which move data from a register to memory).

3. The instruction formats are few in number, with all instructions typically being one size.

RISC architectures include MIPS, ARM, PowerPC, PA-RISC, SPARC, and the now-defunct Alpha. However, if MIPS and ARM own the embedded market, the most powerful computers today appear to be hybrids of Intel chips and Nvidia chips. Why this is so is a long story.

Terminology

____

Early descriptions of processor microarchitecture emphasized the distinction between (active) control circuitry and (passive) datapath circuitry that just followed orders. As time went on, many control functions were incorporated into the datapath. Even so, the conceptual need for distinct control circuitry remained because while we can imagine an intelligent datapath that can analyze and control aspects of executing individual instructions, only separate control circuitry can maintain a more global picture of how different instructions should interact.

Moreover, all datapaths soon became pipelined. Today, it is more natural to speak of an instruction-execution pipeline with some control functionality that makes use of various on-chip hardware resources and is ultimately guided by control circuitry that is _notionally_ outside the pipeline.

RISC pipelines were immediately successful, yet they did not stand still.

Rather, they continued to evolve as heavier and heavier performance demands were placed on them. This continued until the first RISC inflection point in the 1990s. In fact, the RISC micro-architecture dominant in the 21st century is radically different from the one dominant in the 20th century (the multilevel cache was the troublemaker). RISC 1.0 has in-order instruction-execution pipelines: machine instructions are executed in _program order_. RISC 2.0 has out-of-order instruction-execution pipelines: machine instructions are executed _out of program order_ using dataflow ideas to overcome the strict von Neumann control flow in RISC 1.0. We start with the simpler RISC 1.0.

RISC Instruction Execution

____

We now study the MIPS RISC instruction-execution pipeline (at least in the form that was popular in the 1980s). We waste no time on nonpipelined datapaths. First, we need the concept of _pipeline_.

Consider a computer system that takes in operations on the left, computes them, and then pushes out results on the right. In a pipeline, we may push in new operations on the left long before getting the results of previous operations pushed out on the right. A pipeline is characterized by three parameters: First, there is the peak input bandwidth (the maximum input rate the pipeline will tolerate). Second, there is the operation latency (the time for an operation to complete). Third, there is the pipeline occupancy (the number of uncompleted operations in the pipeline at any one time).

Pipelines take time to reach their equilibrium state. We may feed operations into an empty pipeline but need to wait until we get our first result. It is only after the pipeline has reached its equilibrium state that the result bandwidth on the right will exactly match the input bandwidth on the left.

Here is a picture:

```
                          pipeline

                     +-----------------+
                -->  |  -->  -->  -->  |  -->
bandwidth = 2        | concurrency = 6 |        bandwidth = 2
                -->  |  -->  -->  -->  |  -->
                     +-----------------+

                       latency = 3
```

Suppose initially the pipeline is empty, and we apply a sustained input bandwidth of two operations per cycle. Eventually, the pipeline will reach equilibrium, and we will receive a sustained output of two results per cycle. In the picture, it looks like we are supplying peak input bandwidth, but I never said this. In general (Little's Law), at equilibrium, the occupancy (concurrency) of the pipeline is the latency-bandwidth product.

Pipelining has performance consequences. In the picture above, system throughput (result bandwidth) is 2 results per cycle. In contrast, if we were to wait for the previous result(s) before supplying new input, throughput would drop to 2/3 result per cycle. In general, pipelining a system multiplies the throughput by the latency.

The MIPS 'fdxmw' instruction-execution pipeline is a special case of the general pipeline sketched above. At equilibrium, the input bandwidth is 1, the output bandwidth is 1, the latency is 5, and the occupancy (concurrency) is 5.

The MIPS pipeline is divided into five stages (or "boxes"), each of which performs a well-defined task. Here is a picture:

```
+--------------------------------------------------------------------------+ P
|     <I-cache>               <Register file>                 <D-cache>    | r
|                                                                          | o
|                             <Control circuitry>                          | c
|              +-+           +-+           +-+           +-+               | e
|              | |           | |           | |           | |               | s
|     <f-box>  | |  <d-box>  | |  <x-box>  | |  <m-box>  | |  <w-box>      | s
|              +-+           +-+           +-+           +-+               | o
|              f/d           d/x           x/m           m/w               | r
+--------------------------------------------------------------------------+
```

In addition to the square boxes (shown using angular brackets and box names), which you may think of as combinational logic (this is a slight exaggeration), there are rectangular boxes, called "latches", which you may think of as sequential logic (actually, they are merely collections of state elements).

There is a flow of data, from left to right, through boxes and latches. Less happily, there is bidirectional data flow, at great cost in time and energy, between certain pipeline boxes and the hardware resources pictured at the top of the processor. Remember the time and energy required to move data!

1. The f-box reads the memory address of the next instruction from the PC register inside the f/d latch and fetches it from memory. It then updates PC by adding 4. Each box must complete its work, including all latching, in one clock cycle. The I-cache intercepts the memory request and, if it has a copy of the instruction, delivers the instruction to the f-box. Thus, 32 bits of address travel up to the I-cache, and 32 bits of instruction travel down to the f-box. Before the cycle completes, the f-box latches the fetched instruction in the f/d latch. Note that the f-box pokes the I-cache with a memory byte address, which is nothing other than an _array index_; after all, memory is an array of bytes.

2. The d-box has multiple tasks. It must decode the instruction, noting the operands (register and immediate), and the destination register (if any). It also localizes any register operands in the instruction from the register file.

Register names travel up to the register file, and register values travel down to the d-box, which latches them in the d/x latch. Note that the d-box pokes the register file with one or more register names, which are _array indices_, if we view the register file as an array of registers. No instruction has room to store register values; the registor fields in an instruction contain register designators.

The d-box also processes conditional branches. Suppose the branch is 'bne r1,r2,loop'. The d-box tests inequality of the two registers it has localized. If they are not equal, the branch is deemed taken, and the d-box adds the offset (a multiple of the immediate 'loop') to the base register PC, thus causing the branch to take place on the next successful fetch. With 16-bit immediate, we can jump 2^15 instructions up or down.

Again, all these tasks must be completed within a single clock cycle. In particular, the branch-target address must be written into PC before the cycle completes, if the branch is taken. Many of the names/values isolated during decoding are passed down the pipeline as data for use by other boxes.

3. The x-box is actually a case statement, which obviously requires control.

i) In a memory reference, we add the base register and the offset to compute the memory address.

ii) In a register-register instruction, we perform the operation specified by the opcode.

iii) In a register-immediate instruction, we perform the operation specified by the opcode, using the immediate as an operand value.

Do x-boxes exist? Of course, there are no such things as ALUs; instead, there are various functional units, perhaps an integer adder, an integer multiplier, a floating-point adder, a floating-point multiplier, a shifter, a logic-operation unit, etc., etc. So the x-box begins to seem somewhat ghostly if you think about it too carefully.

Also, although the x-box is a case statement, it is not really the one we have described. Consider 'l.d f6,-24(r2)', 'add r1,r2,r3', and 'addi r1,r1,8'. As far as the x-box is concerned, these are all _one case_. The x-box's job here is simply to perform an integer addition; where these integers may have come from is irrelevant. In contrast, 'mul.d f0,f2,f4' is a different case, since now the x-box needs to perform a floating-point multiplication.

4. The m-box is another case statement. If the instruction is a load, the m-box reads from memory, and latches the result in the m/w latch. If the instruction is a store, the m-box writes to memory taking the value to be stored from some pipeline latch. Otherwise, the m-box does nothing. Of course, data and control must sometimes flow from the x-box to the w-box even if the m-box itself does nothing. This is called "bypassing". Note that the D-cache stands between the m-box and the real memory. The D-cache also intercepts memory requests.

5. The w-box does the same thing if the instruction is a load, or an ALU instruction, which necessarily produces a result. Namely, it takes the loaded value, or the ALU result, which must both be in some pipeline register, and writes it in the destination register in the register file.

Not all instructions use all five boxes. A conditional branch uses only the f-box and the d-box. A store uses f, d, x, and m. An ALU instruction uses f, d, x, and w. Only a load uses f, d, x, m, and w.

Information Flow

____

Consider the x-, m-, and w-boxes. Here is a brief description of the data flow. If the x-box is to act on some value or values, it must receive it (or them). Also, it may receive a further data value that specifies more precisely the action to be performed (e.g., the number of bits to shift).

In a store instruction, the m-box receives a data value from the pipeline, namely, the value to be stored. It also receives a data value that is the memory address of where in memory the former value is to be stored. In a load instruction, the m-box only receives the memory address. When the instruction contains a destination register, the w-box receives both a data value, and a register name from the pipeline. This is what is to be written back and where it is to be written back to. Obviously, one only feeds data to a box when one intends that box to do something.

The control signals are too many to permit a brief description. We therefore invent a set of toy control signals that give us some of the flavor of control, without getting us bogged down in unnecessary details. The toy control signals are 'no-op', 'add', 'multiply', 'shift', 'load', 'store', and 'write back'. If any box receives a box-appropriate control signal at the start of cycle 'c', then it will perform the specified action during cycle 'c'. The 'no-op' signal means "do nothing".

Reality Check

____

Consider the f-box. Allegedly, it sends PC to memory and fetches the next instruction. Although we don't call this a _memory reference_, i.e., a load or a store, it certainly is a _memory access_. How many processor cycles does it take to access memory on a typical computer? In lecture 1, I suggested the figure of 200 cycles. You cannot address memory and get back a word in a single cycle. What is going on here?

In reality, the f-box sends PC to the _instruction cache, or _I-cache_. A cache is a complex state element that can store copies of some, but obviously not all, memory locations. Most caches (L1$, L2$, L3$) are small enough to fit on the processor chip. So, we have to pretend two things that are not true. First, that it is possible to access the I-cache in a single cycle. Second, that the I-cache magically _always_ has a copy of just the instruction we need. Surprisingly, for reasons explained later, these two white lies do not grossly misrepresent the actual performance of the I-cache.

Hint: this good fortune has something to do with the fetch-execute cycle.

Consider the m-box. Allegedly, it either sends a data address to memory and receives a data value (this is a load), or else it sends a data address plus a data value to memory, thus depositing the data value (this is a store).

Recall that the m-box only does work when the pipeline is executing a memory reference. Still, each memory reference is certainly a memory access. Do we have a problem here?

In reality, the m-box interacts with the _data cache_, or _D-cache_. This is another complex state element that can store copies of some, but obviously not all, memory locations. Again, we pretend two things that are not true. First, that it is possible to access the D-cache in a single cycle. Second, that the D-cache magically _always_ has a copy of the data value we wish to load or store. For data, these two white lies _can_ cause serious problems. It depends on the memory-accessing pattern.

Consider program P executing during some interval I. Suppose that, during this interval, program P only uses data from, say, 20K distinct memory locations. We say that these 20K locations constitute the _working set_ of program P during interval I. If program P's working set fits into the computer's cache, then we can't really say we have a problem. However, if the working set doesn't fit into the D-cache, then we _may_ have a problem: we may repeatedly fail to find the data copy we want in the D-cache, and thus may repeatedly be forced to access the actual memory, at much greater cost.

Which of these two cases is the real one depends on the memory-accessing pattern of program P. Some patterns generate enormous working sets that are too big to fit into any D-cache.

This problem even has a name: it is officially called the _Memory Wall_.

Consider Moore's Law. It says that processor performance, which we may roughly model as the product of the number of logic transistors times the clock frequency in Hz, increases exponentially over time. Before 2003, both factors increased exponentially. Now, we have to be very careful with the clock frequency. Still, replacing a single power-inefficient core with many power-efficient cores increases the power efficiency of the processor chip as a whole, and thereby allows us to increase performance without overstepping our power budget. We don't ask that each factor in Moore's Law increases exponentially; we only ask that their _product_ increases exponentially. Moore's Law, interpreted in this fashion, is alive and well in the multicore era, with one important qualification. Update: In 2020, Moore's Law is far from being alive and well.

Increased arithmetic performance is only possible if there is increased delivery of data operands to functional units. Memories are making some improvements to memory latency and memory bandwidth, but not fast enough to keep pace with processor improvements. Raw processor performance increases faster than raw memory performance. An ideal, or perfect, cache could easily compensate for the mismatch between processor demand and memory supply.

However, in the real world, caches are _not_ ideal, and programs are not always _cache friendly_. In the worst case, program performance may be limited by memory performance, and be unaffected by increases in processor performance. All moderately educated computer professionals agree that the Memory Wall and the Power Wall are the two main challenges that must be overcome by today's computer designers. Dealing with the Power Wall is mandatory. What about the Memory Wall? We could accept to only write programs that play nicely with today's caches. But this concession might be selling our birthright for a mess of pottage.

Pipelining

____

Pipelining is a naturally efficient way for a datapath to execute machine instructions. It was standard in vector supercomputers. The basic idea is

to allow different workstations ("boxes") to work on different instructions at the same time. By overlapping the execution of different machine instructions, we can make significant improvements in the pipeline's execution

throughput.

Say that executing some machine instruction is a task 'T'. Suppose we break 'T' into a _sequence_ of nonoverlapping subtasks: 'T = t1; t2; ...; tn'. Now, we provide a specialized workstation for each subtask (these are our boxes). Think of the processor clock as ringing a bell at the start of each processor cycle. For simplicity, let's stick with having five stages, with the f, d, x, m, and w boxes as our specialized workstations. An instruction pipeline doesn't have to have five stages---it could have 21---but this was the design of an early RISC processor.

When the bell rings, each workstation passes the partially executed instruction to the workstation on its right.

A picture may help:

```
         1  2  3  4  5  6  7  8  9  0  ...  clock cycle
instr 1  f  d  x  m  w
instr 2     f  d  x  m  w
instr 3        f  d  x  m  w
instr 4           f  d  x  m  w
instr 5              f  d  x  m  w
instr 6                 f  d  x  m  w
instr 7                    f  d  x  m  w
...                           ...
```

Look at clock cycle 5. We have five boxes working on five different instructions.

Is this advantageous? Suppose each box completes its work in one clock cycle. Therefore, the time to execute an instruction (the instruction _latency_) is 5 cycles. But when the pipeline achieves cruising speed, a new instruction completes every cycle, giving the pipeline an execution _bandwidth_ of one instruction per cycle. (We call this _single-cycle pipelining_). The ideal speedup when a pipeline is pipelined is thus equal to the pipeline _depth_ (the number of stages).

Each of our five boxes is a _combinational_ circuit, but the clocked pipeline as a whole is a _sequential_ circuit. To make this work, each box is followed immediately on its right by a set of (nonISA) _pipeline registers_, which is

called a "pipeline latch". The basic requirement is this: Prior to the end of a clock cycle, all the results from a given stage must be stored in the pipeline latch to its right, in order that these values can be preserved across clock cycles, and used as inputs to the next stage at the start of the next clock cycle.

Also, it is reasonable to think of the boxes as combinational circuits that compute Boolean functions, and of the latches as finite state machines that provide control of boxes---among other things.

Exercise: Pick a cycle and a box and describe the control plus data received at the beginning of that cycle.

Space-time Diagram

____

The figure above is a space-time diagram. These diagrams show the evolution of the pipeline in time and are useful in capturing aspects of the control and data flow. Let me repeat the figure with some actual (but not very interesting) sets of instructions.

```
                1  2  3  4  5  6  7  8  9
add r1,r2,r3    f  d  x  m  w
add r2,r3,r4       f  d  x  m  w
add r3,r4,r5          f  d  x  m  w
add r4,r5,r6             f  d  x  m  w
add r5,r6,r7                f  d  x  m  w
```

The instruction-execution pipeline reaches equilibrium in cycle 5, when all five boxes are (notionally) active, and five distinct instructions are being processed at the same time. At equilibrium, one instruction is fetched per cycle, one result is produced per cycle, the latency is 5 cycles, and the pipeline occupancy (concurrency) is 5, which is the latency-bandwidth product.

In this toy program, no data flows between instructions, which are all independent. As described earlier, data does flow between different boxes working on the same instruction. For example, in the first instruction, the x-box receives the values of 'r2' and 'r3', which it adds. The w-box receives the register name "r1" and the sum computed by the x-box. However, the m-box receives no data at all. Since an add is not a memory reference, there is no work from the m-box to do. Instead, it performs a no-op. I sometimes write "n" in place of "m" when I want to emphasize that the m-box is inactive. The data sent to the w-box hops right over the m-box.

In this diagram, what control signals do the boxes receive? Consider cycle 1. The f-box receives the control signal 'fetch', while _all_ the other boxes receive the control signal 'no-op'. By reading the columns of the diagram, one can see which boxes receive activating control signals, and which boxes are instructed to do nothing. (Qualification follows).

The exception to this rule is the m-box. Although I have written "m" in five columns, in fact, the m-box receives the 'no-op' control signal in _each and every_ cycle (not just the columns in which "m" appears).

In every cycle, the f- and d-boxes receive either the 'no-op' control signal or the 'fetch' and 'decode' control signals, respectively. From a control standpoint, they are not very interesting case statements. For this reason, we focus on the x-, m-, and w-boxes.

When the x-box does something, it may be any one of a range of arithmetic or logical operations. When the m-box does something, it is either a load or a store. When the w-box does something, it is always written back.

A pipeline is a machine with a control system. We will see more interesting space-time diagrams when we learn about pipeline dynamics. But a space-time diagram by itself shows the pipeline's kinematics. For all boxes other than the m-box, if a box name appears in the column corresponding to cycle 'c', then that box is active in cycle 'c'. However, if the m-box's name appears, then it may or may not be active.