A 0.6 μm BiCMOS Processor With Dynamic Execution

Robert P. Colwell & Randy L. Steck
Intel Corporation

Agenda

- P6 Status
- Implementation
- System
- Test & Validation

This tour was derived from a paper presented at the International Solid State Circuits Conference (ISSCC), Feb. 1995. It is a guided technical tour through key elements and methods in the P6 processor design and describes how we achieved twice the performance of the Pentium® Processor on the same manufacturing process.

The tour covers the following key sections:

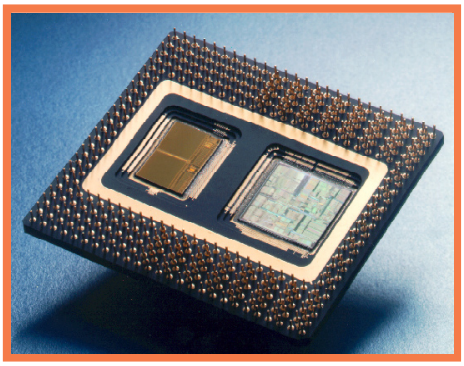
1.Technology and status of the processor.

2.Discussion of implementation techniques with the help of the microarchitecture block diagram, the processor pipeline and some of the circuit techniques that comprise the CPU.

3.Insight into P6 at a system level and why we invested effort, not just in the CPU and the cache, but in the rest of the system as well.

4.And finally, a few words on testing and validation.

Status



- Estimated 200 SpecInt92 @ 133MHz
- 0.6 µm BiCMOS
- 5.5M transistors
- 691x691 mil (306 sq.mm)
- 2.9V Vdd
- 20W peak

The first thing to notice is that there is something different about the first P6 processor: there are actually two die in the package. The one on the left is the L2 cache. The one on the right is the CPU. These die are wire bonded into the same package allowing a full speed interface to the L2.

This combination of two die in a package yields an estimated 200 SPECint92 at 133 MHz. We use the word "estimated," because to be compliant with SpecMarks the public must be able to purchase the system. This is not a presentation about a product -- P6 is not a product yet. Instead, we are presenting the technology of the P6 processor.

We are manufacturing this part on a 0.6 micron BiCMOS process, a mature Intel process also used for the 100 MHz Pentium(R) processor. The CPU has 5.5 million transistors, with a die size of approximately 691 mils on a side or 306 square millimeters. It runs at 2.9 volts and dissipates a peak of 20 watts. Typical power dissipation is 14 watts.

Technology profile

- 32-bit Intel Architecture processor
- Dynamic Execution microarchitecture
  - Out-of-order
  - Speculative Execution
  - Superscalar
  - Superpipelined
  - Micro-dataflow
- 8K/8KB non-blocking L1
- 256KB integrated non-blocking L2

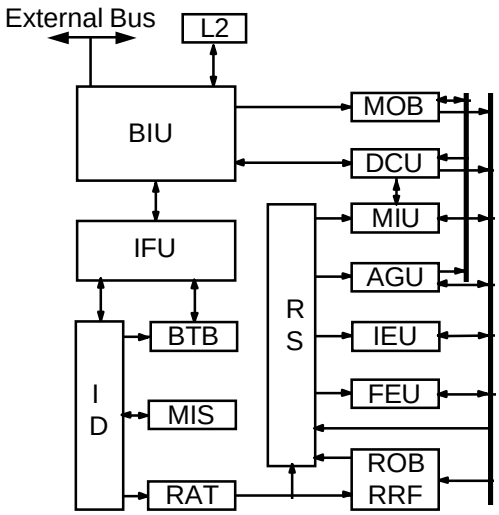
A few other pieces of context will help you in understanding what we are about to cover. The first is that this is a true 32-bit Intel Architecture processor. It is not some strange 64-bit device from outer space that had been rumored in the trade press for the last couple of years. This is binary compatible with all previous Intel architecture processors.

The next bullet you see above has the catch phrase "Dynamic Execution." It is a phrase that we coined to stand for a number of other words that are hard to remember all by themselves, including out-of-order, speculative execution, superscalar, superpipelined, all wrapped around a dynamic dataflow engine at the core. You'll see a lot more details on the dataflow engine later.

The L1 cache is 8K/8K split, very similar to the Pentium(R) processor, with one crucial difference: our cache is non-blocking. This is important in an out-of-order engine because otherwise an access that missed in the cache, which it could have made speculatively, would stall all the accesses behind it. The potential for performance boost would be lost, so we made P6 processor's L1 cache non-blocking.

The L2, which as you saw was in the package, is 256K bytes. It also is non-blocking. If you take a miss on it, the access is "parked" and other cache accesses can be made. The L2 can support four concurrent cache misses before store buffers are used.

Implementation: Microarchitecture



Let's take a look at the block diagram of the P6 processor to see how it fundamentally works. The processor is organized in three sections:

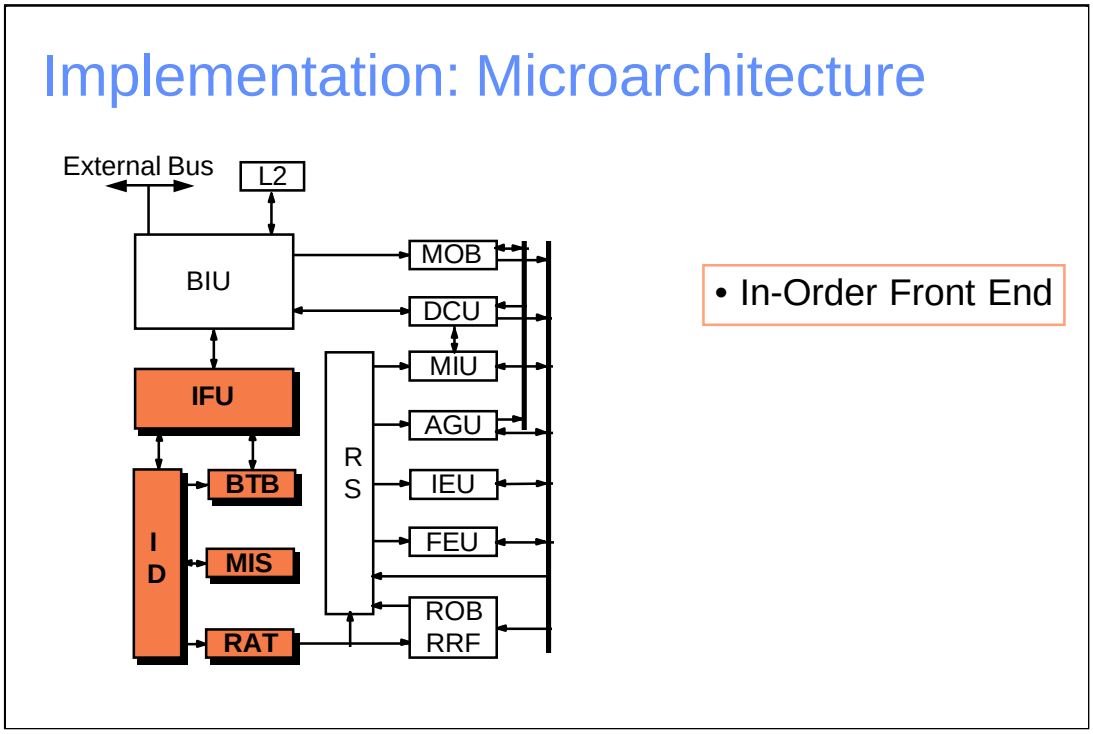
In-order front end

Out-of-order core

In-order back end

The two in-order sections have to be there in order to make this machine have the same program semantics as an Intel 486(TM) processor. Our job in designing the P6 Processor was to make it look to software as if there was an extremely fast Intel 486 processor hidden inside the box. The way to achieve that is to make it look like an Intel 486 processor to the memory subsystem on both ends. We used many innovative architecture techniques inside to create a high performance machine, but these are all transparent to application programs.

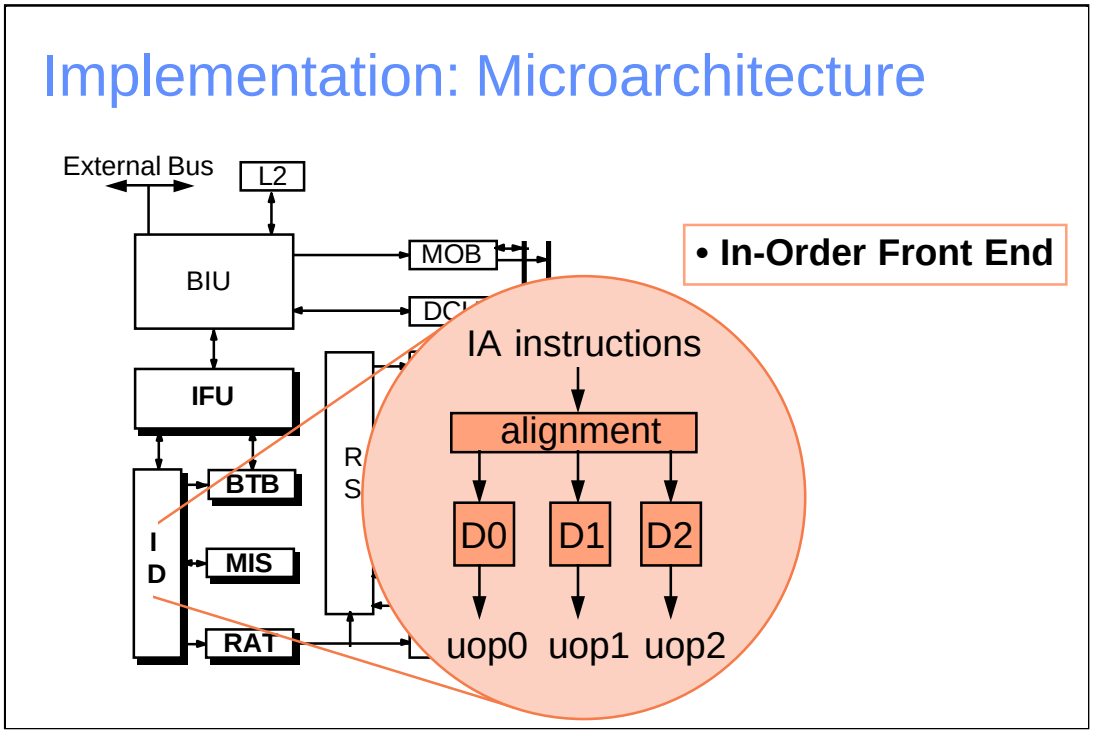
We'll walk through those three pieces one by one.



The in-order front end consists of:

IFU (Instruction Fetch Unit)
BTB (Branch Target Buffer)
ID (Instruction Decoder)
MIS (Micro Instruction Sequencer)
RAT (Register Alias Table)

The IFU, which contains the Icache, is the place where the Intel architecture instructions that constitute the programlive. What the IFU does is supply a line's worth of information to the decoder. The Icache knows where to fetch those bytes from because the branch target buffer guesses and tells it where to look. The branch target buffer is a non-trivial design using a 512 entry two-level adaptive algorithm. You will see why we thought it was appropriate to do a more elaborate BTB when we get to pipelining.



Microarchitecture: Instruction Decode

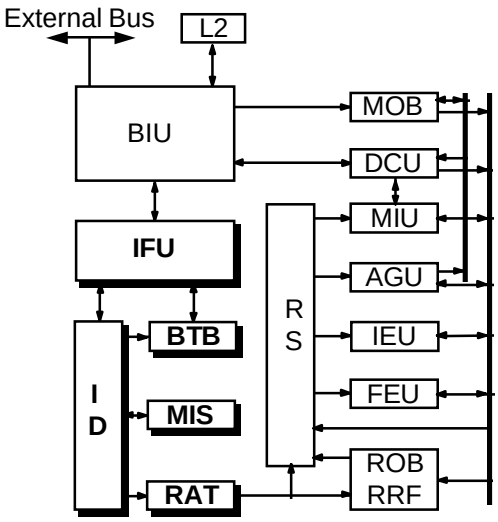
The instruction decoder's job is to break the Intel architecture instructions down to micro-ops. Micro-ops are the atomic unit of work in the P6 processor and are comprised of an opcode, two source and one destination operand. These micro-ops are fixed length and are more general than the Pentium(R) processor's microcode since they need to be scheduled. At the bottom of this diagram you can see there are three micro-ops generated per clock. This makes the P6 a superscalar processor of degree 3.

The decoder takes the instructions from the IFU. They're getting aligned because the variable nature of the encoding of an Intel architecture instruction is such that it is difficult to tell where instructions start and end. Then there are three separate decoders, one for each aligned instruction that we will map into the micro operations. These micro operations flow out the bottom.

The three decoders on the diagram look identical, but actually they are not. One of the decoders is capable of translating any Intel architecture instruction into the constituent micro-ops that we will execute in a P6. Any Intel architecture instruction that can be mapped into four or fewer micro-ops will be directly translated by this decoder. More complex instructions will be used as indices into the microcode instruction sequencer (MIS) which will issue the appropriate stream of micro-ops. The other two decoders are simpler. This is an attempt to match the die size and area complexity to the actual performance characteristics of normal code. If the two less capable decoders see an instruction that they are not qualified to decode, they will pass that instruction to the more capable one.

So, the thing to remember from this diagram is that three micro-ops per clock cycle come flowing out of the decoder.

Implementation: Microarchitecture

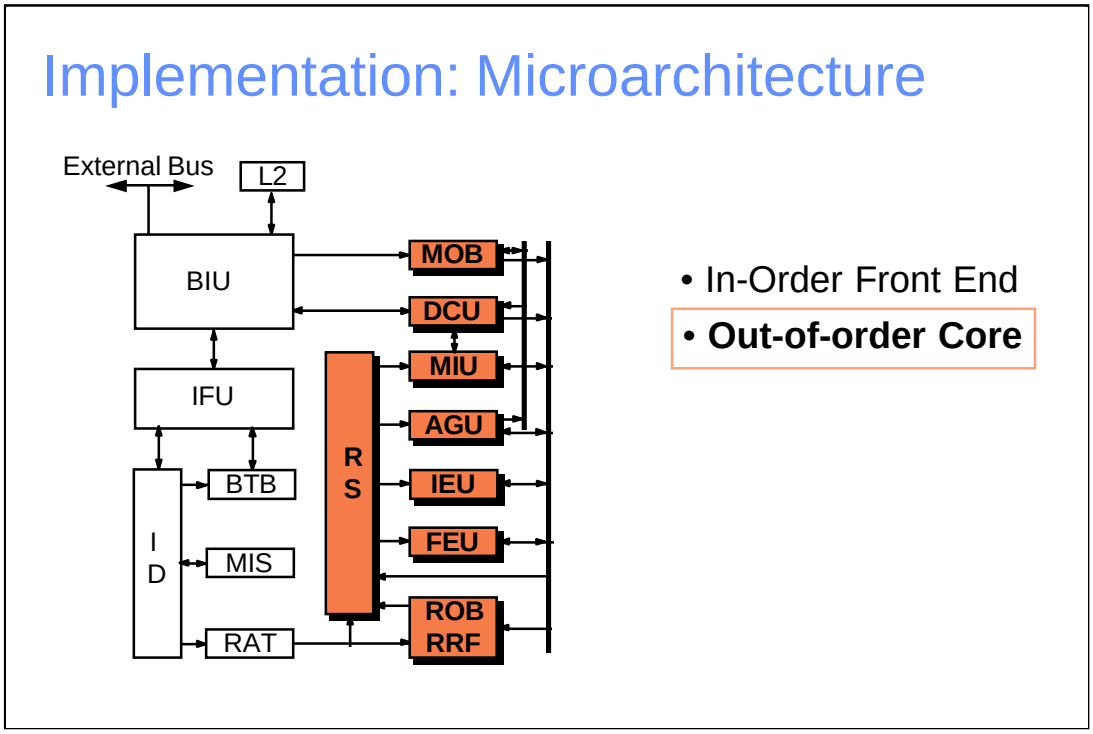


- In-Order Front End

Microarchitecture: Register Renaming

The micro-ops exiting the instruction decoder go to the Register Alias Table (RAT). The Intel architecture does not include a large register set. This is unfortunate in an out-of-order machine, because it can lead to unnecessary delays on register reuse. So, we rename the logical registers specified by the program source to physical registers that reside in the reorder buffer (ROB), which we will see in a later diagram.

This is the end of the in-order part of the processor.



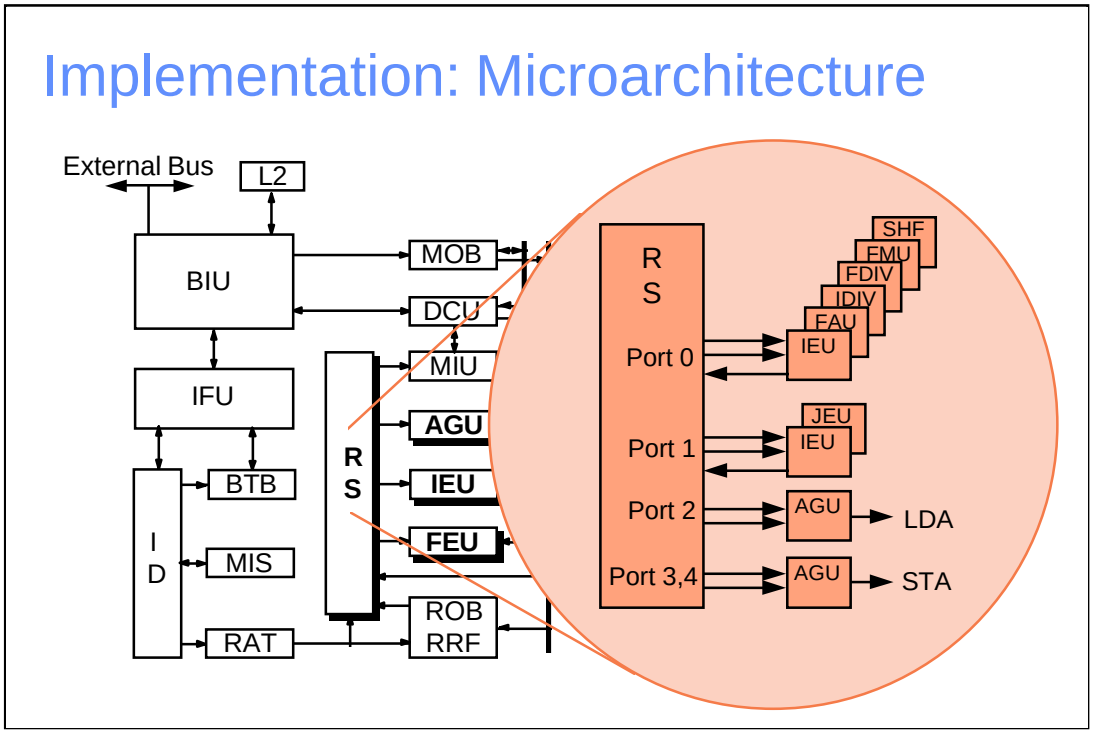
Microarchitecture: Out-of-Order Core

In the previous diagram the micro-ops were flowing out of the decoder and heading for the out-of-order core of the machine. If you look carefully at the bottom center of the picture, you'll see the micro-ops actually go to two different places. One place is the reservation station (RS), and the other is the reorder buffer (ROB).

The micro-ops have to go to the reorder buffer because we must have an in-order back end. In order to reimpose program order on the micro-ops later we have to know what order they belong in. When the micro-ops flow into the reorder buffer they effectively take a place in line so that we can remember how to retire them later and keep the right program semantics.

Micro-ops also go to the reservation station so they can be sent to the actual execution units. There are 20 entries in the reservation station, and each micro-op will take up one slot in that unit. The reservation station is not a generic FIFO or simple data structure; it is more generalized. Each entry can handle integer, floating point, flags--anything that is "renamable" and can be part of the execution of the machine.

It appears from this picture that there is one integer execution unit, one floating unit, one address generation unit (AGU), etc. That is because the picture was too cluttered to show the actual structure, as you see in the next diagram.



Microarchitecture: Execution Core (RS and EUs)

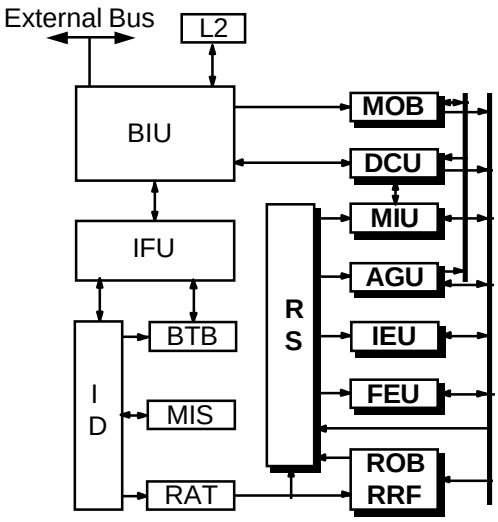
The reservation station (RS) actually has five ports. You will notice on Port 0 that there are several units attached to it: integer, floating point adder, integer divide, floating point divide, floating point multiply and a shifter. Why is all that hanging off one port? The main reason is that the floating point units need a wider data path. The intermediate data form of a floating point value is 86 bits wide. This requires two operands and one result -- that's a lot of bits. Rather than reproducing that on every port, we balanced the processor by putting the floating point units on Port 0, shared with some integer EU's, and more integer capabilities on Port 1. Port 1 has an integer execution unit (EU) and the jump execution unit.

Ports 2, 3, and 4 are dedicated to memory accesses. Port 2 generates the load addresses through the address generation unit (AGU). Ports 3 and 4 generate the store addresses through their own AGU.

Each of these ports has its own writeback path back to the reservation station -- high performance needs very high bandwidth back to the reservation station. It made the picture here too cluttered to show that there is a full cross bar between all those ports so that any returning result could be bypassed to any other unit for the next clock cycle. This is crucial for high performance in back-to-back micro-ops.

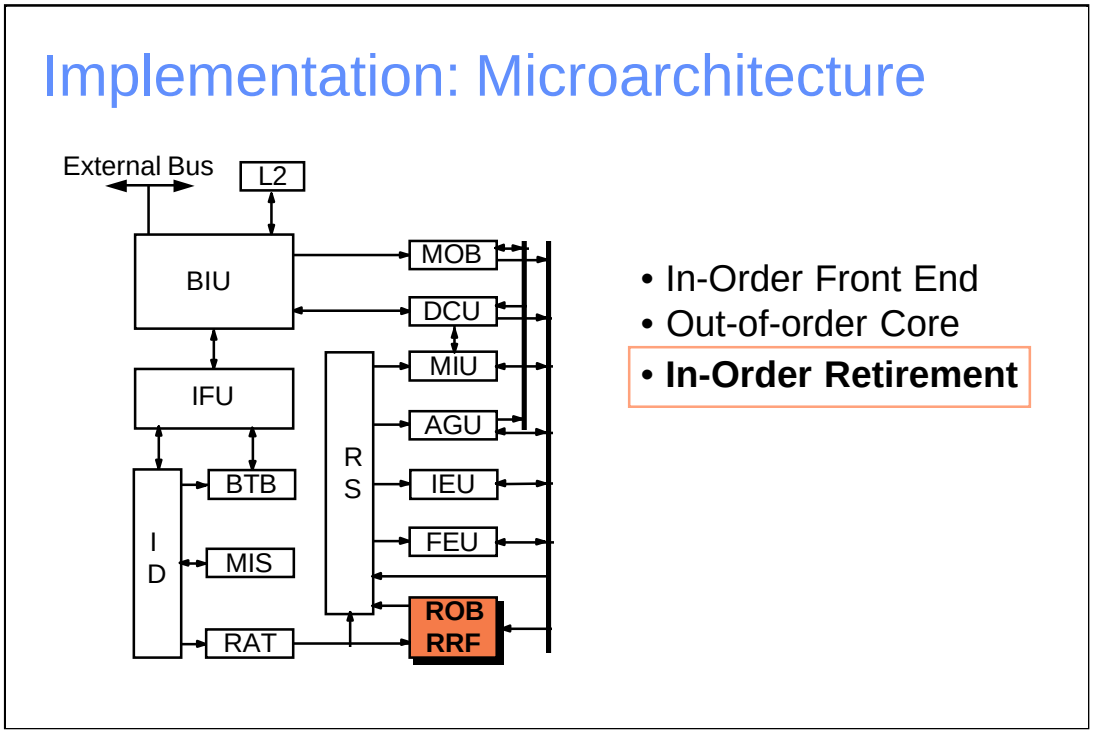
Once the reservation station has picked a micro-op to send out to an execution unit, it goes out on the appropriate port, it executes, and the result comes back. The result goes to two different places. One is back to the reservation station. Why? Because there may be other micro-ops in that reservation station waiting for that very piece of data before they themselves become data-ready and able to execute.

Implementation: Microarchitecture



- In-Order Front End
- **Out-of-order Core**

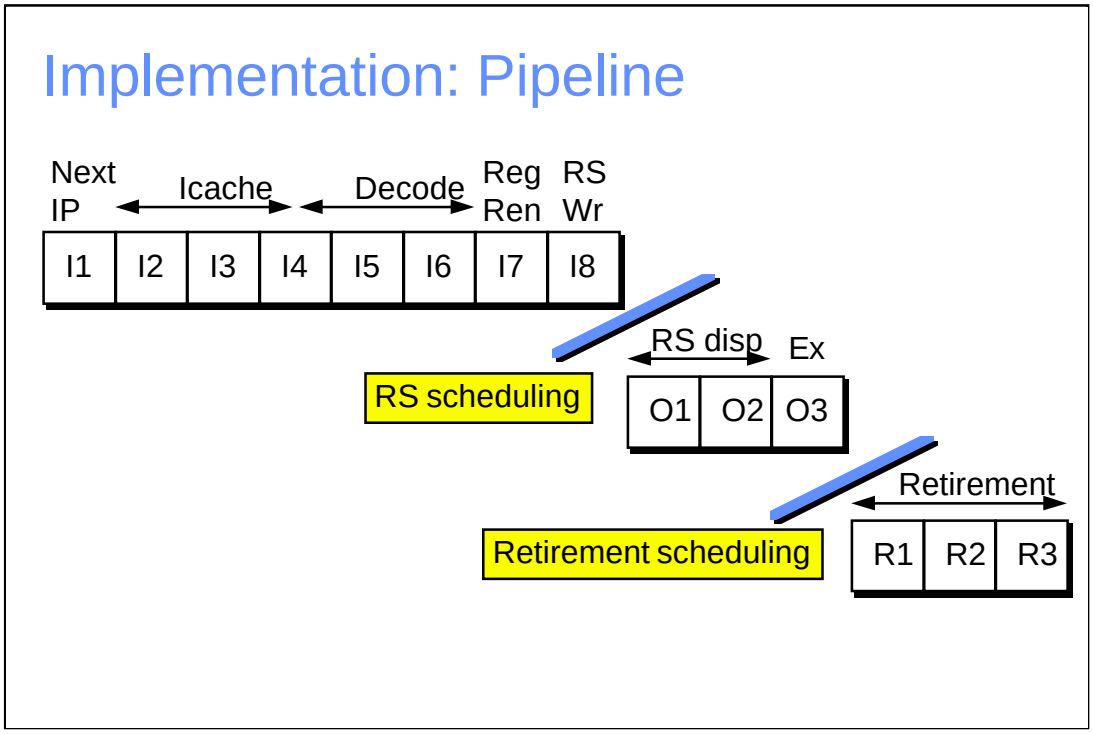
The other place is the reorder buffer. It determines which micro-ops at any given moment are capable of being retired. They are capable of being retired only if they have actually executed, they have all the results ready, it is their turn in line and there is no other thing pending. We will return to retirement momentarily, but the important thing is that when the micro-op has actually executed, the results go back to the reorder buffer so that the ROB knows that it is completed.



Microarchitecture: In-order Retirement

By retirement we are referring to the act of taking the speculative state in the machine and permanently and irrevocably committing it to permanent machine state elsewhere in the machine. What may not have been obvious is that most of this machine is speculative. At any given moment you can flush it all and not lose program correctness. You do not want to do that very often, of course, because for performance reasons you want to keep the machine busy. Mispredicted branches, interrupts, breakpoints, traps and faults can cause some or all of the speculative state to be flushed.

The in-order retirement process is the act of not changing your mind any more. When a micro-op has executed and the ROB knows it is from the path of certain retirement and it is that micro-op's turn, it is retired. Retirement means taking data that was speculatively created and writing it into the retirement register file (RRF). If you looked inside the RRF, you would actually find entries like EAX or Floating Stack Entry #3, whereas in the reorder buffer there are 40 slots that are generalized (no dedicated EAX).



So now you've seen the basic way that the machine flows, and the way micro-ops flow through the machine. It may have occurred to you that there seems to be a lot of work going on in here, so how deeply pipelined must this machine be?

The flow of micro-ops through the P6 processor is controlled by the super-pipeline shown above. Were you to count from left to right, you would end up with 14, but we don't consider this machine a 14-stage pipeline, because some of these stages overlap almost all the time.

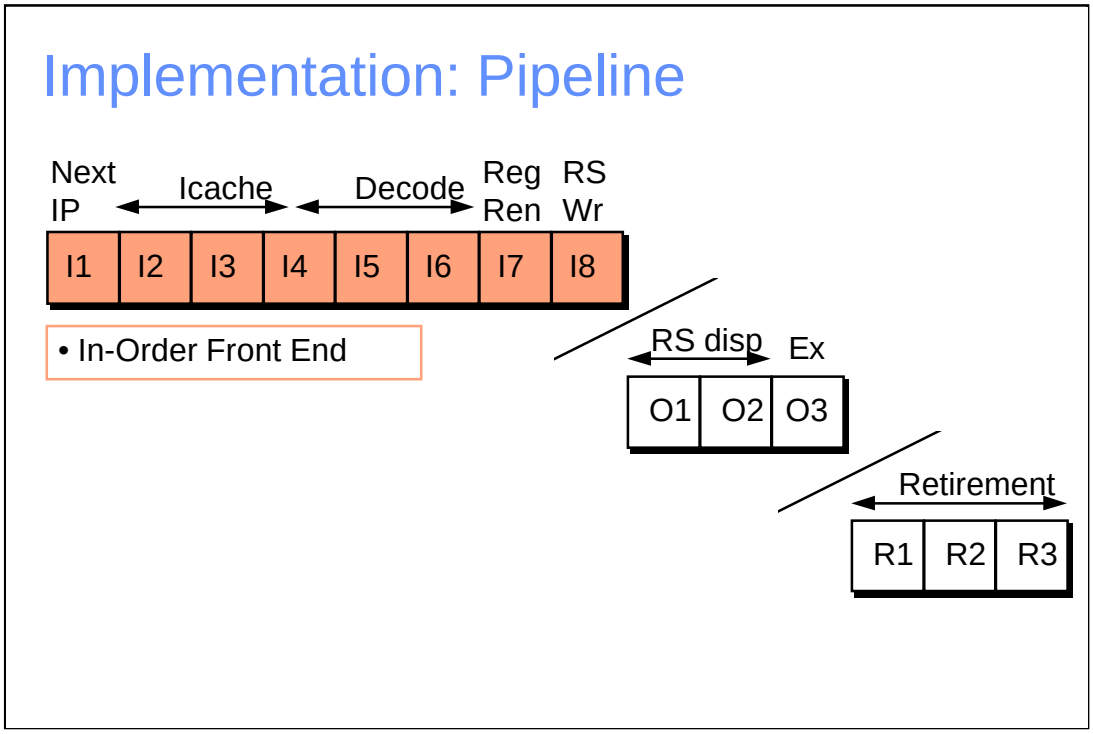
You'll notice one further thing. We do not show it as one long set of boxes because it is not implemented as one big pipeline. The problem with creating one long pipeline is that the aggregate pipeline would have to run at the speed of the slowest stage. So we segmented the pipeline into the three pieces you see here. They correspond to the three pieces you've already seen:

Pipeline of in-order front end

Pipeline of out-of-order core

Pipeline of in-order retirement

The diagonal slashes are intended to represent the queuing effect. There is some time that goes by between the pipe segments as micro-ops flow from one pipe segment into the next.

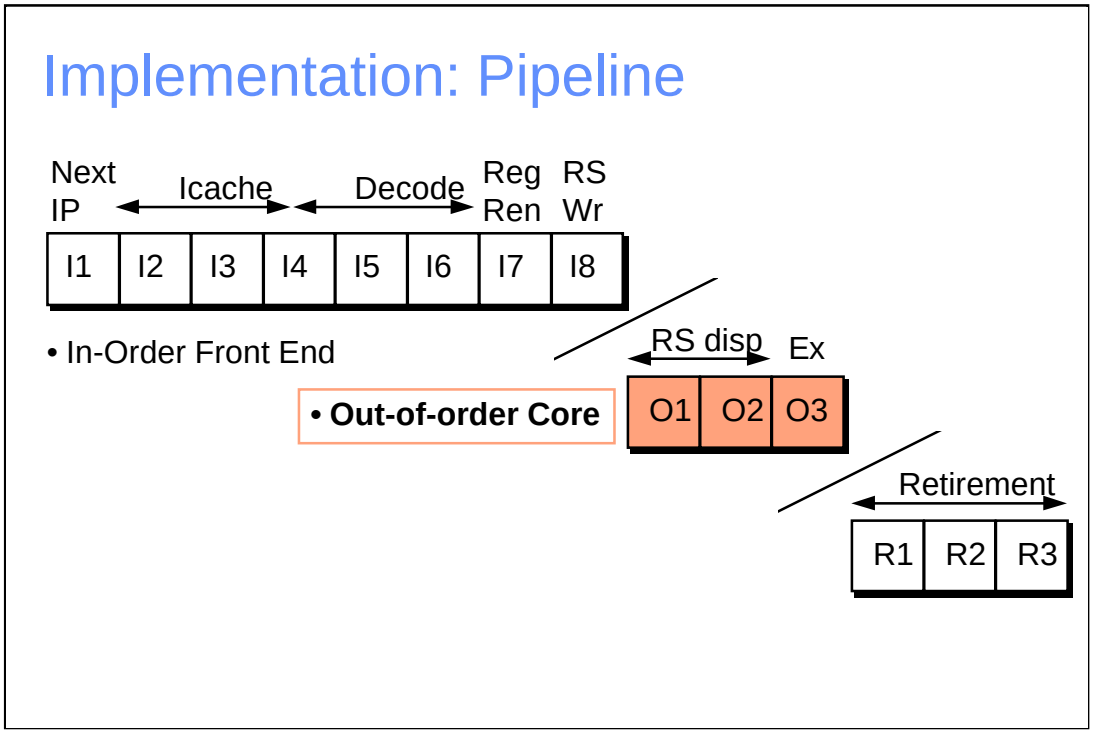


Processor Pipeline: In-order Front End

The in-order front end involves eight clock cycles. The first one identifies the next instruction pointer (IP). As you recall, that is the branch target buffer deciding where is the best place to look in the Icache for the next cache line.

The next 2-1/2 clock cycles are the Icache access. The next 2-1/2 after that are the decode, including the three decoders and instruction alignment. The clock cycle after that is the register rename, which is at the end of the in-order pipeline.

Finally, the reservation station write cycle can usually be overlapped with at least one of the clock cycles in the next pipeline segment.



Processor Pipeline: Out-of-Order Core

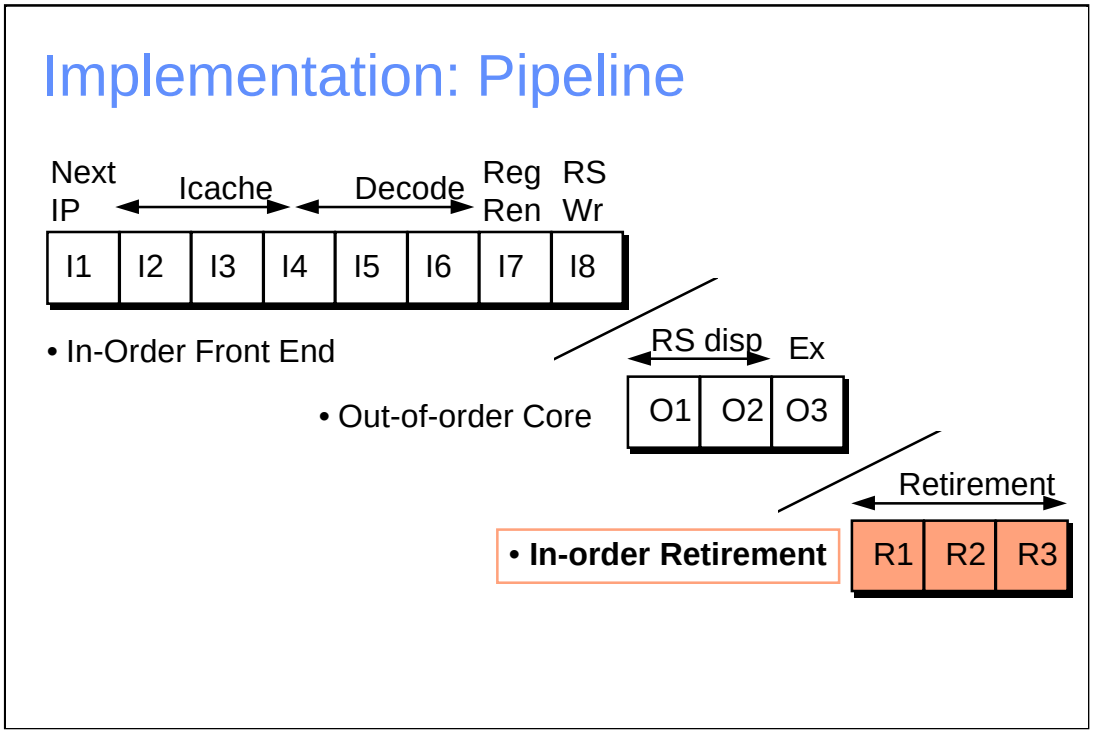
Next we enter the out-of-order core, which as you might remember was the entry into the reservation station and the writing into the ROB.

For an integer op, say in a register-to-register add, that is a one-cycle execution in this machine.

So it is shown as three cycles here, two cycles for the reservation station to correctly identify which micro-ops have all the operands and are ready to go, and then one cycle for the actual execution and the return of the results.

There are other pipeline segments not shown, for example the memory subsystem or floating point, which have deeper latency. For floating point the execution state would have stretched out several additional cycles. The memory subsystem is an entirely different system and the pipeline is much more difficult than this one, so we did not use it as an example.

Once an execution unit has created its result, the result flows back to the reservation station to enable future micro-ops and also flows down into the reorder buffer to enable retirement.



Processor Pipeline: In-order Retirement

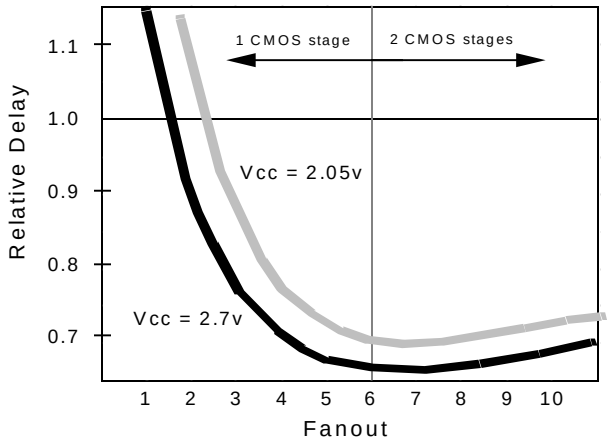
The retirement process actually takes three clock cycles. Part of that is trying to make sure that we only retire things as a group. For example, any given Intel architecture instruction may in fact map into one micro-op or it could map into several.

The retirement process has to make sure that if you retire any micro-ops of an instruction you retire all of them automatically. Otherwise the machine would have inconsistent state if it happened to take an interrupt at the wrong moment. So, the retirement process has a fair number of edge conditions to make sure it gets it right. That is why it takes extra clock cycles.

Now you've seen how the pipeline basically works and how many clock cycles are in it. This is why one of the technical phrases included in Dynamic Execution is "Superpipelined." This superpipelining allows us to push the clock rate as high as possible.

We mentioned earlier that the BTB was a much more complicated, capable design than in previous processors. The reason is the deep superpipelining - with this many clock stages, it pays to fill the pipe with the right work to do.

Implementation: BiCMOS



Circuits: BiCMOS

Now for some of the circuit implementation techniques.

There is a nominal relative line of 1.0 about 3/4 of the way up this graph. It shows what CMOS alone would have achieved in terms of speed path delay vs. fanout.

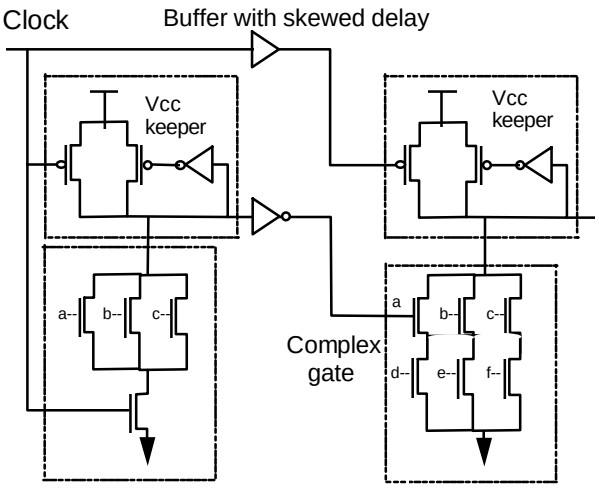
But we did not use only CMOS in this machine; we used BiCMOS, and the question this graph is addressing is "how much did the bipolar aspect of this affect us in this design?"

What you see is that with fanout across the x axis, anything above 2 or 3 loads bipolar suddenly wins big in our methodology, especially when you get up to fanout of 5 or 6 where CMOS would have to go to two stages to do the buffering. We are at about a 0.7 speed path delay of normal CMOS. In other words, it is 30% faster! This continues for even higher fan outs.

One of the standard concerns people raise to using BiCMOS is if the Vcc goes too low, the efficiency of the process goes down and you lose the effect. The second gray line above the first one is our response to that. That is 2.05 volts on Vcc and it is still 30% faster than the CMOS was. In other words, this was a win for us.

Across the board, we estimate that using bipolar was responsible for about 15% of our overall clock speed.

Implementation:
Delayed Precharge Domino



Circuits: Delayed Precharge Domino

This diagram looks like standard domino logic, except for the additions to the right. For example, there are two delays in between the two stages. The thing to notice is on the left it looks more or less like standard domino logic, where you have a clock in series with everything at the bottom of the picture, a Vcc keeper precharge section at the top and logic down at the bottom.

There are two basic hazards with standard domino logic. One is the logic hazard: you have to carefully manage the precharge time vs. the gating time. If you get them on at the same time you are in danger of corrupting the data and losing the hold time to the next stage. The other hazard is that precharge and the logic can in fact look like they are in series and a timing error could create a flow through path for power.

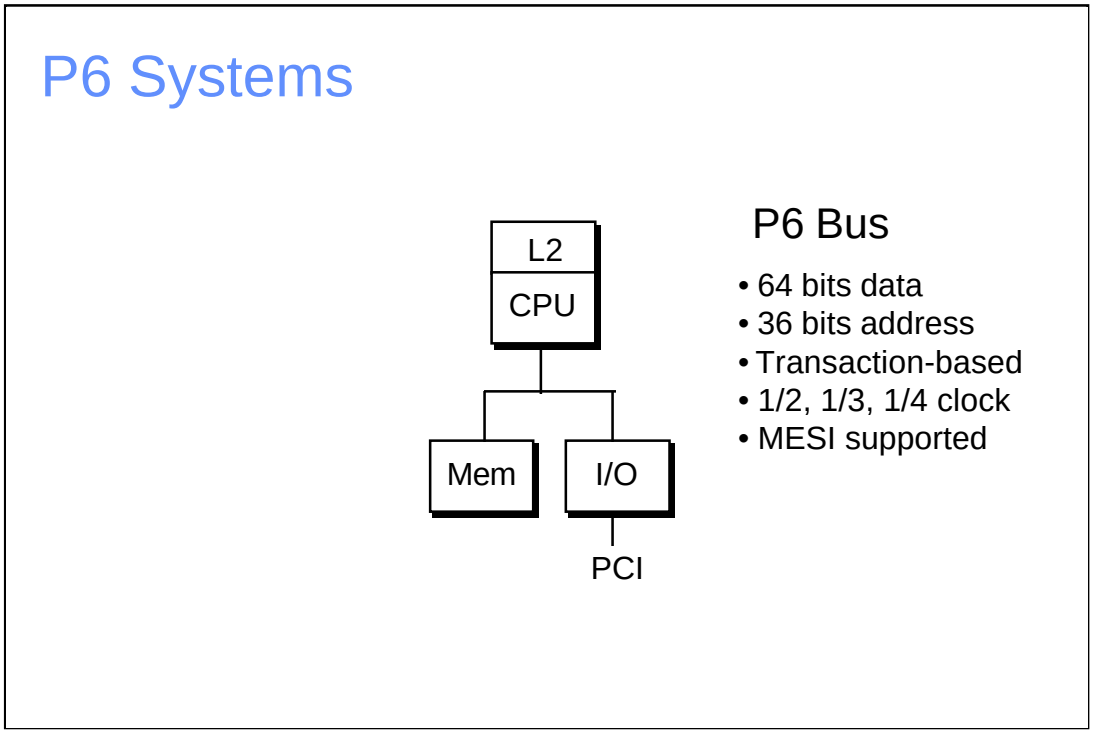
To avoid the hazards we included timing delays in between the two stages. We tuned these delays to make sure that these two hazards were both addressed at the same time. By carefully staging when the precharge occurred on the next stage, we carefully kept it away from the place that the logic was being done. This solved both the problems.

Agenda

- P6 Status
- Implementation
- **System**
- Test & Validation

You've now seen some implementation details, the micro architecture block diagram, the pipelining, and some of the circuit techniques.

Let's take our attention to a higher level now. Where does this design fit in a systes environment?

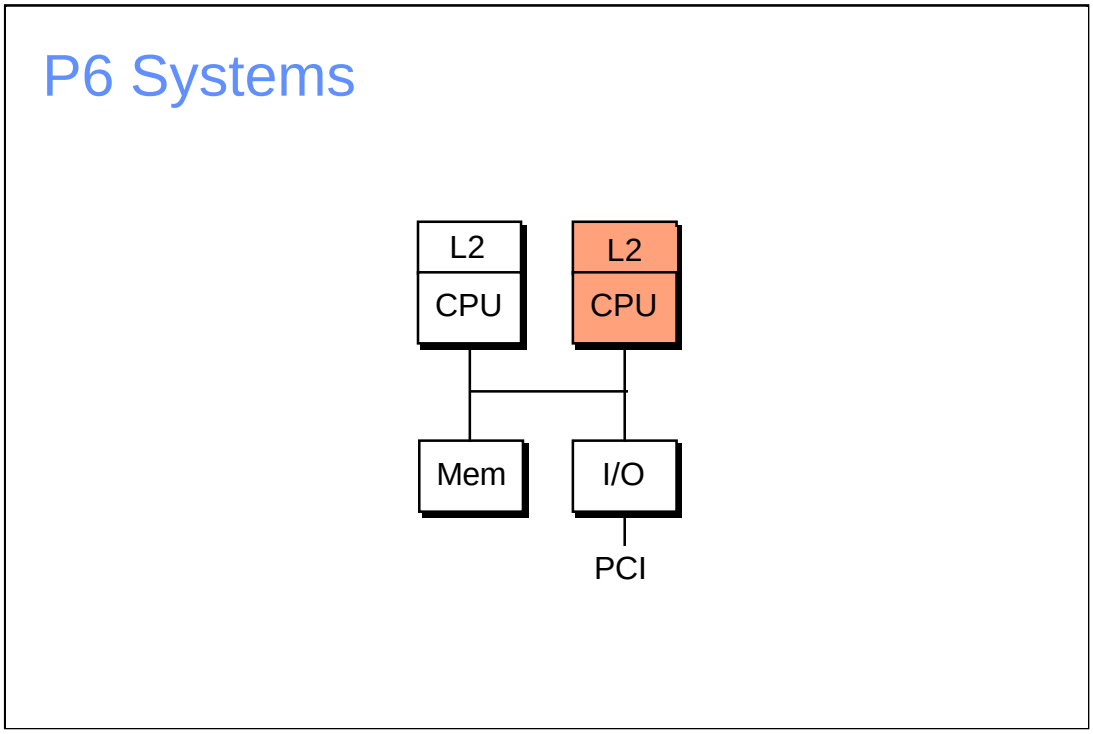


P6 System Bus

Let's take our attention to a higher level now. Where does this design fit in a systems environment? This diagram looks essentially like a standard block diagram of a computer system except for the L2, which is in the same package as the CPU.

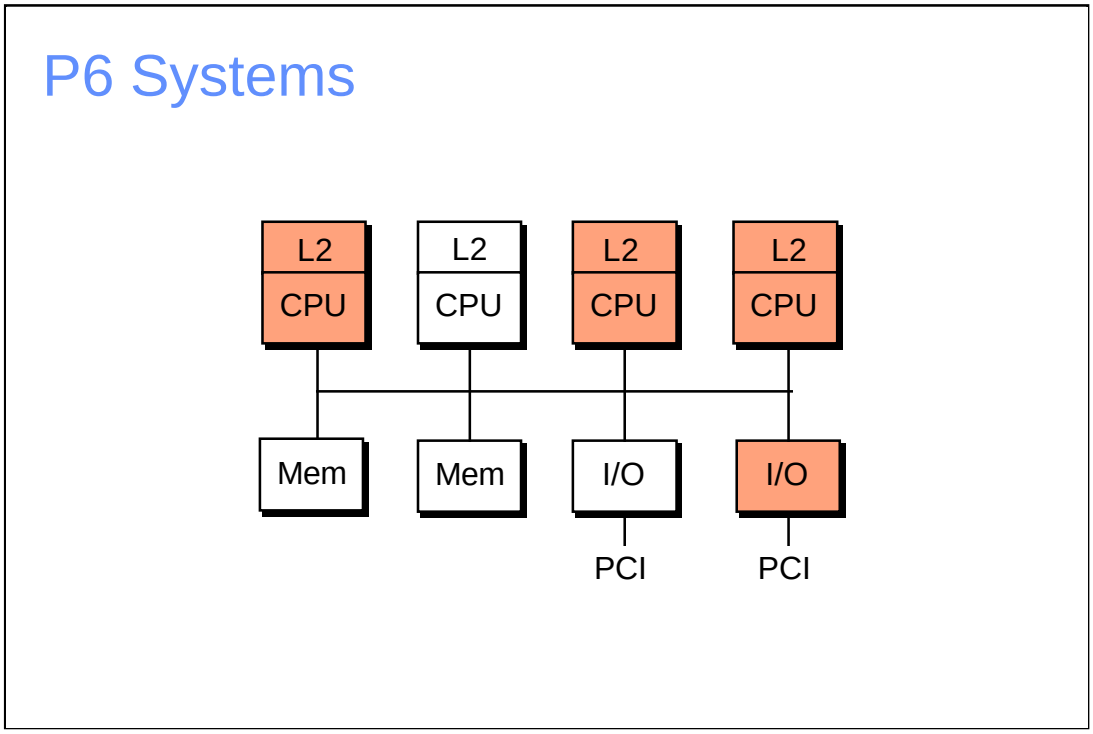
The first thing to note is that when we got to this point in designing the P6, we saw how to design a fast CPU, but if we left the rest of the machine alone we would have taken a relatively balanced system and unbalanced a piece of it. The problem with an unbalanced system is you can't predict the performance. And generally it has unpleasant surprises in other ways. So we set out to explore what other things needed attention.

That resulted in the following bus: The P6 bus is a 64-bit data bus. It has 36 bits of physical addressing. It's transaction based which means that any access that's looking for data gets on the bus with the request and gets off the bus until the data is coming back. In the meantime, other agents on the bus can use that bandwidth. It runs at 1/2, 1/3, or 1/4 of the CPU clock speed and it has snooping support built in for multiprocessing.



Glueless Multi-Processor Environment

Once we had solved the bus problem, we were just a short step away from glueless MP. All that was lacking was some initialization protocol and a few details about handling interrupts. In a P6 system, all it takes to do MP is to take the wires from one CPU, stretch them sideways and put another socket down. The P6 does all the rest of the work. The question of how far you can stretch this expansion is answered on the next diagram.



Complete P6 MP System

The P6 processor's glueless multiprocessing model allows expansion up to four processors. If you do use four P6 processors, however, the net aggregate requirement for bandwidth is such that the system should include more memory and more IOs. We made provisions for those as well.

The diagram shows a fully expanded P6 system. The thing that is interesting about it is that it is extremely high power and high performance, but resides in a very small box. It is unprecedented in its capability.

Test & Validation

- Billions of simulation cycles
- Correct by design
  - Code assertions
  - New coverage tools
  - Coding conventions (signal names)
- Simulation included chipset +4 CPUs
- Extensive testability & debug hooks

Test and Validation

Now a few words about testing and validation. How did we ensure the best possible functionality in a processor as complicated as the P6?

We allocated a lot of our resources and people to this issue, including several teams working on the validation of the CPU, the L2 and all the members of the chipset plus the bus, running billions of simulation cycles. We paid a lot of attention to a concept we called "correct by design." Correct by design does not mean we can prove there are no errata. Any machine more complicated than a hammer has errata in it. However, we paid tremendous attention to making sure that we got the design right up front--it is far better to avoid an errata than to have to find it later. Correct by design in our case meant, for example, the usage of code assertions, which is a technique that operating system vendors, compiler writers and people who designed large complex systems have found useful in the past. It means the designers can capture the specific knowledge they had about some boundary conditions inside the unit when they were writing the Register Transfer Level (RTL) code. This allows them to say, "When you get to this part of the code, X had never better equal Y. That would cause my circuit to malfunction." They don't have to say why--they just have to capture that in one line. From then on, all the validators who don't know about that corner case can use this knowledge. If that corner case ever becomes not true, a flag pops up and it alerts the validators that there is an error. They do not know what it is, but they know there is an error. This is extremely useful for the same reasons that it is useful in compilers and operating systems.

<more on next slide>

Test & Validation (continued)

- Billions of simulation cycles
- Correct by design
  - Code assertions
  - New coverage tools
  - Coding conventions (signal names)
- Simulation included chipset +4 CPUs
- Extensive testability & debug hooks

New coverage tools: We wrote, used and deployed extensively a set of coverage tools in the validation of our part. These coverage tools told us which sections of the overall design were being very well tested and which ones needed extra help. This allowed us to steer our efforts toward the places that needed the help--a much more efficient way to use the people and the resources.

Another thing we did was to use coding conventions. It allowed us to look at statements in the RTL and notice that different clock stages are being combined logically. It also drew our attention to places where there might be an error.

Many of the simulations that we did included not just the CPU but 2-3-4 CPU's plus the chipset, including the bus. This was extremely useful. In fact, the efficiency of the approach was proven when we got first silicon back on all of the chipset, plus the L2, plus the CPU and it all worked together.

The CPU also included extensive testability and debug hooks. Testability hooks basically mean that any feature could be disabled while maintaining correct functionality--allowing reconfiguration for testing. If nothing else this gave us a strong clue as to what the problem might be if there was indeed a problem. In the best case it allowed us to work around other problems in ways we had not conceived of during the design phase.

The P6 processor has approximately two times as many transistors as the Pentium(R) processor. We ran 15 times as many pre-silicon simulation cycles to ensure that errata were minimized as much as possible.

Summary

- Current P6 Status
- Complete Implementation
- Balanced System
- Thorough Testing and Validation

So in summary, you've seen the current status and that there's a complete implementation involved. You also saw the block diagram, the pipelining and some of the circuit techniques. We discussed the need for a balanced system and the requirement for the same, and you saw how met those requirements by improving not just the CPU but the L2, the chipset, and the bus. And finally we discussed how we've given the P6 the most thorough testing and validation to ensure it delivers the best possible functionality.