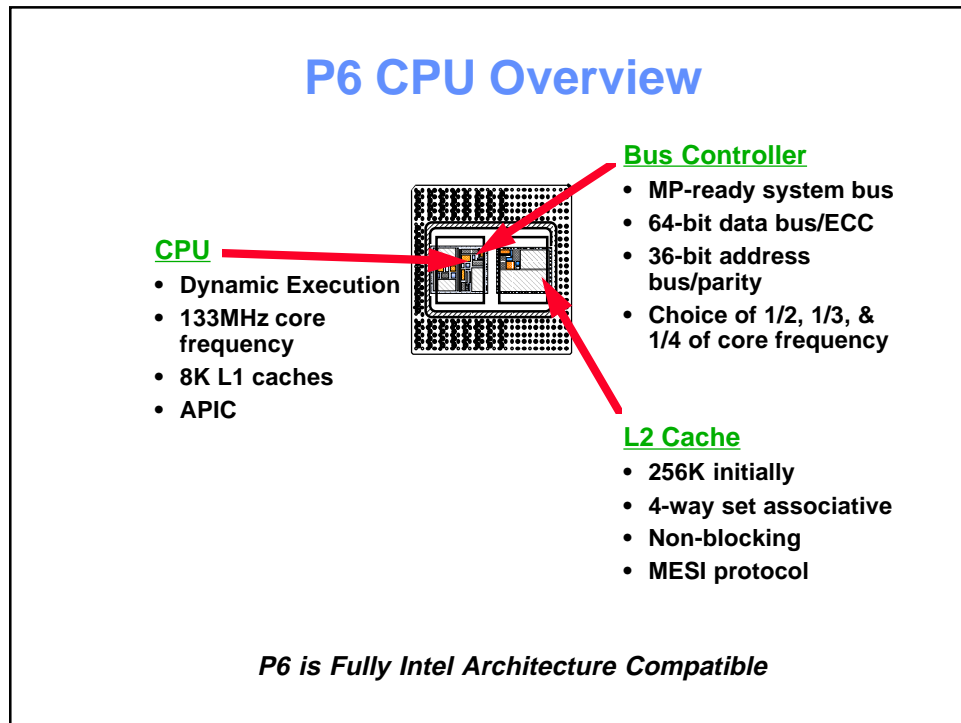


The P6 Architecture: Background Information for Developers

©1995, Intel Corporation



This is an overview of the P6 processor, highlighting all of the important features -- intended especially for software developers.

The first P6 will run at 133MHz. Internal level 1 data and code caches are 8KBytes backed up by the large 256KByte level 2 cache (in the first P6) that is only 3 processor clock cycles longer in access latency. That's faster than ADS# can be asserted on the bus.

P6 is built to allow scalable multi-processing, including a built-in interrupt controller and a multiprocessing-capable bus.

An important feature to note is that the P6 is fully compatible with the instruction set of the Intel486™ and Pentium® processors. P6 will enhance the instruction set, but backwards compatibility is a prime goal.

P6 Dynamic Execution Architecture

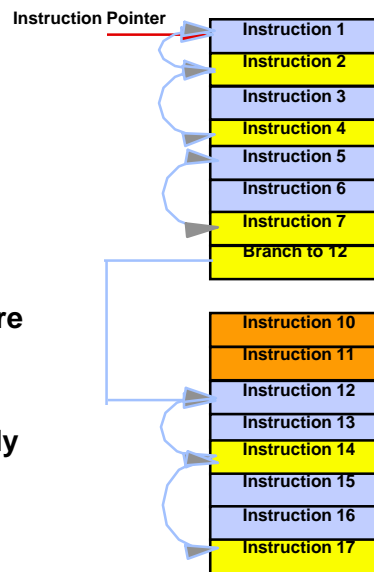
Extends the Intel Architecture Beyond Superscalar

- **Non-blocking architecture**
 - Prevents processor stalls during cache, memory, I/O accesses
- **Out-of-order execution**
 - Executes all available instructions as data dependencies are resolved
- **Speculative execution with multiple branch prediction**
 - Eliminates CPU stalls when branch occurs
 - Further improves effectiveness of O-O-O
- **Register Renaming**
 - Eliminates false dependencies caused by limited register set size
 - Also improves effectiveness of O-O-O

The P6's prime difference over the Pentium® processor is the Dynamic Execution internal architecture. The P6 is designed to be completely non-blocking. To achieve this, P6 integrates the concepts of speculative execution across multiple basic block boundaries (branches) with register renaming and out-of-order execution via a true dataflow compute engine.

Data Flow Engine

- **P6 Implements a dataflow engine**
- **Instructions usually have numerous dependencies (registers, flags, memory, etc.)**
- **Instruction dependencies are rigorously observed**
- **When all sources are available instruction is ready to execute**
- **When execution unit is available, instruction executes**



The P6 implements a true dataflow engine. It will execute any instruction whose dependencies are resolved, potentially well ahead of its “normal” execution time. Even cache misses will not stop it finding useful work to do.

All instructions are executed into a pool of temporary state and “retired” at a later linear time.

We’ll see more about this example set of instructions later.

OUT-OF-ORDER?

- **What is Out-of Order?**
 - Instructions with dependencies resolved, or no dependencies, may execute ahead of their von Neumann Schedule
 - This is a data flow sequence
- **Example on right: a, c: and d: could execute at the same time**
 - This would enable us to execute the sequence in 2 cycles rather than 4
- **In previous Superscalar processors, compilers re-scheduled code to create the same effect. Thus P6 has less dependence on compilers for performance.**

a: $r1 = \text{Mem}(x)$

b: $r3 = r1 + r2$

c: $r5 = 9$

d: $r4 = r4 + 1$

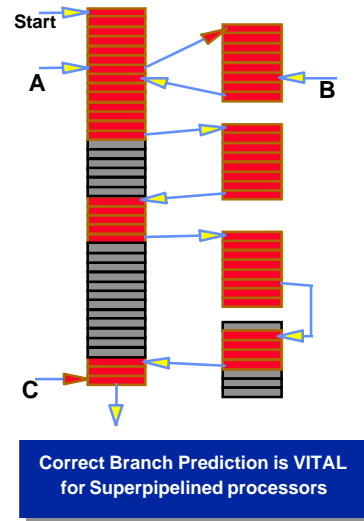
OOO allows the processor to do useful work even though instructions may be blocked!

The simple linear execution of a von Neumann instruction stream is a limiter to execution performance while instructions are waiting for dependencies to be fulfilled. In many code sequences the apparent dependence of instructions upon each other and on processor resources can be removed by allowing the dependence-free instructions to execute ahead of their normal execution time and allowing the processor to hold their results in temporary state.

The example shows four instructions where b,c and d could all execute at the same time with no impact on the correctness of the results. This mechanism is termed Out-Of-Order.

Speculative Prefetch

- Assuming the code sequence on the right:
 - An Intel486™ Processor would have executed from start to A, then flushed the prefetch pipe. It would have resumed pre-fetching and execution at the destination address.
 - A Pentium® Processor could have speculatively prefetched the code from the destination address and as far down as B before the branch at A is actually taken
 - The P6 could have speculatively prefetched the code from start to C and beyond, and can execute any or all of it before the branch at A is actually committed as taken

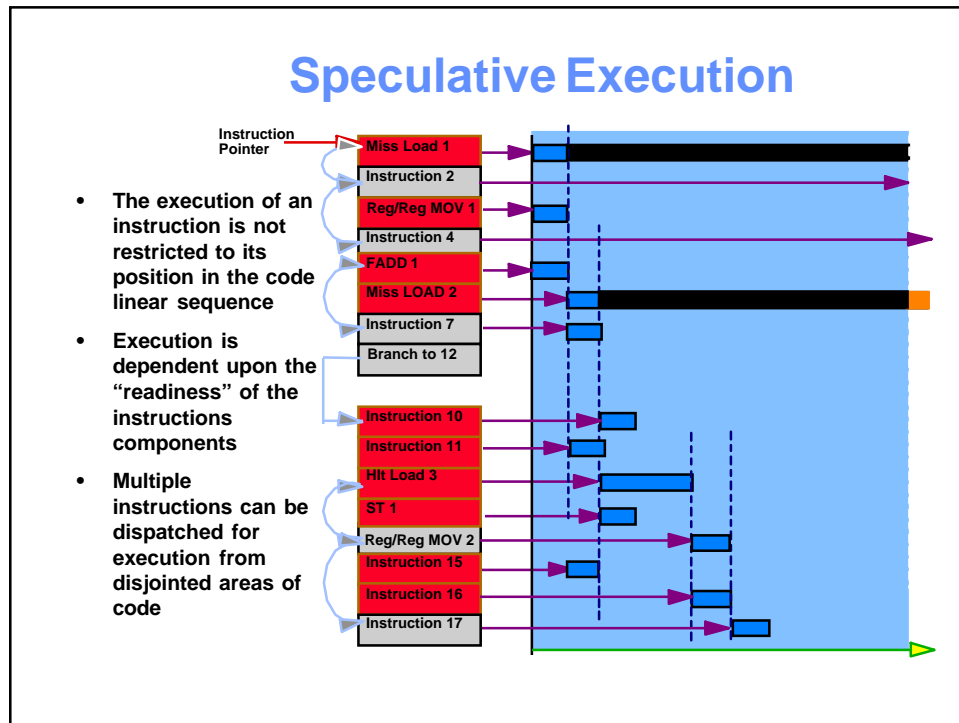


The initial sentence stated speculative execution. To define what we mean by speculative execution we should first look at speculative prefetch. The Intel486™ processor did not implement any level of speculation. The Pentium® processor implemented a degree of speculative prefetch to the point where if a branch was predicted, the code at the destination of the branch would be prefetched potentially as far ahead as the next branch instruction.

The P6 will speculatively predict branches and prefetch the destination code through numerous branches and returns. In the example above, the P6 allows all of the light colored code to be in the P6 internals and to be in various states of execution.

Thus the P6 implements speculative execution as well as speculative prefetch!

Speculative Execution



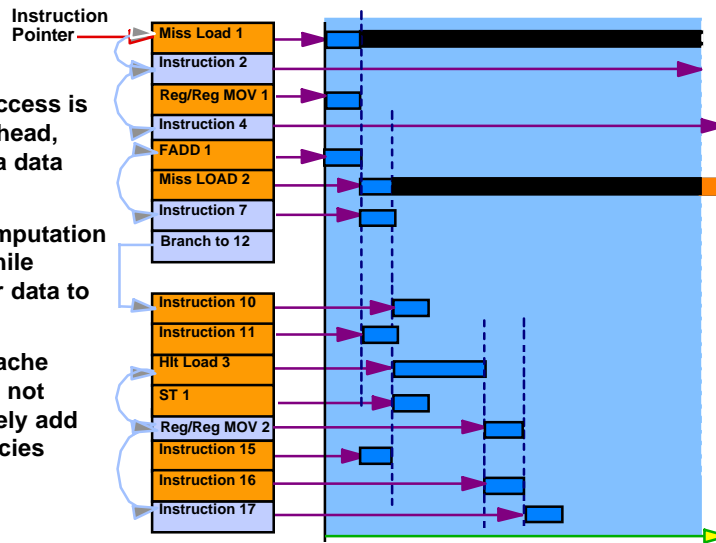
To show what speculative execution actually looks like, the above diagram gives an example. In this code sequence, the gray instructions have dependencies that are initially unresolved (with arrows indicating the instructions they are dependent upon). The orange instructions have no dependencies.

The timeline shows the dispatching of three instructions for execution. The first instruction is a load instruction that misses the cache. The other two instructions are executed in one clock. In the second clock three further instructions are dispatched, the first of which is another cache miss. It can be seen from this diagram that we have actually executed four instructions and started a fifth (the second cache miss load) before the end of the second clock and all well within the latency of the first cache miss .

In a normal sequential, or even supercalar machine, we would still be waiting for the data of the first cache miss to return. The full latency of the second cache miss load is masked by the latency of the first miss, showing only the incremental latency of the of the second miss -- the orange colored section at the end of the timeline.

Non Blocking Architecture

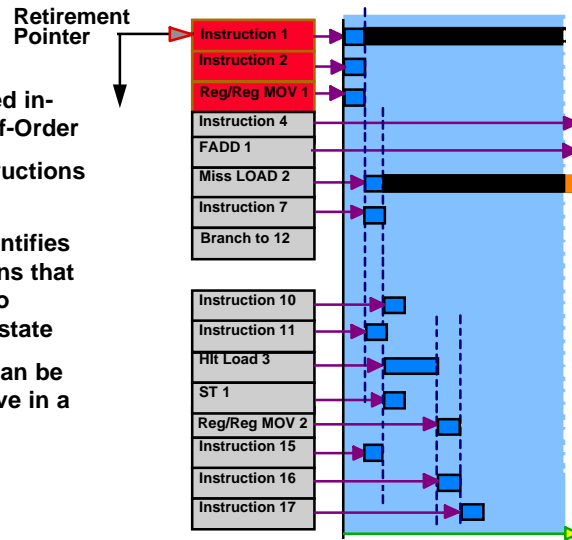
- Memory access is initiated ahead, acting as a data prefetch
- Useful computation is done while waiting for data to return
- Multiple cache misses do not cumulatively add their latencies



We have seen the graphic of this slide before in the context of OOO. This time we need to emphasize the non-blocking aspects of what is shown. The cache miss does not stall the processor. There is useful execution continuing within the cache miss latency and the full latency of subsequent cache misses is hidden.

Instruction Retirement

- Instructions are fetched in-order, executed Out-Of-Order
- The retirement of instructions is in-order
- Retirement pointer identifies the block of instructions that are being committed to permanent processor state
- Multiple instructions can be retired. They always live in a sequential block of 3 instructions

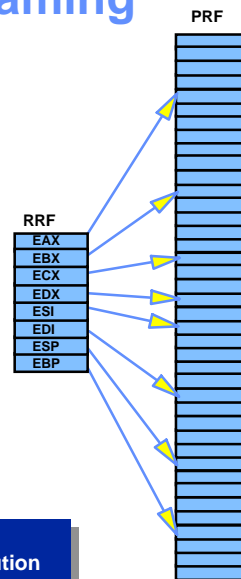


Instructions are dispatched and executed out of order. To allow for real binary compatibility, where not only do instructions mean the same things but already coded algorithms mean the same thing, the P6 implements an in-order update to permanent processor state - Instruction Retirement.

Instructions are retired at a maximum rate of 3 per clock cycle. The ROB looks like a circular sequentially addressed retirement pointer that continues to sequentially retire instructions chasing the speculative execution engine.

Register Renaming

- Intel Architecture processors have a relatively small number of general purpose registers
- False dependencies can be caused by the need to reuse registers for unconnected reasons, i.e.
 - `MOV EAX, 17`
 - `ADD Mem, EAX`
 - `MOV EAX, 3`
 - `ADD EAX, EBX`

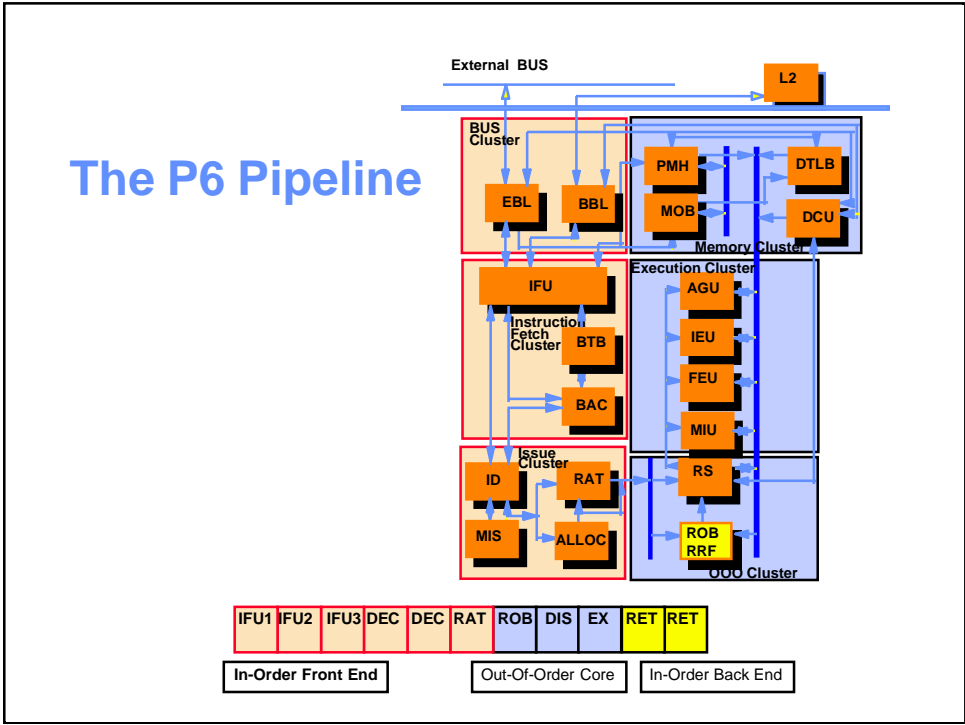


Increased number of registers allows non-blocking architecture to continue execution

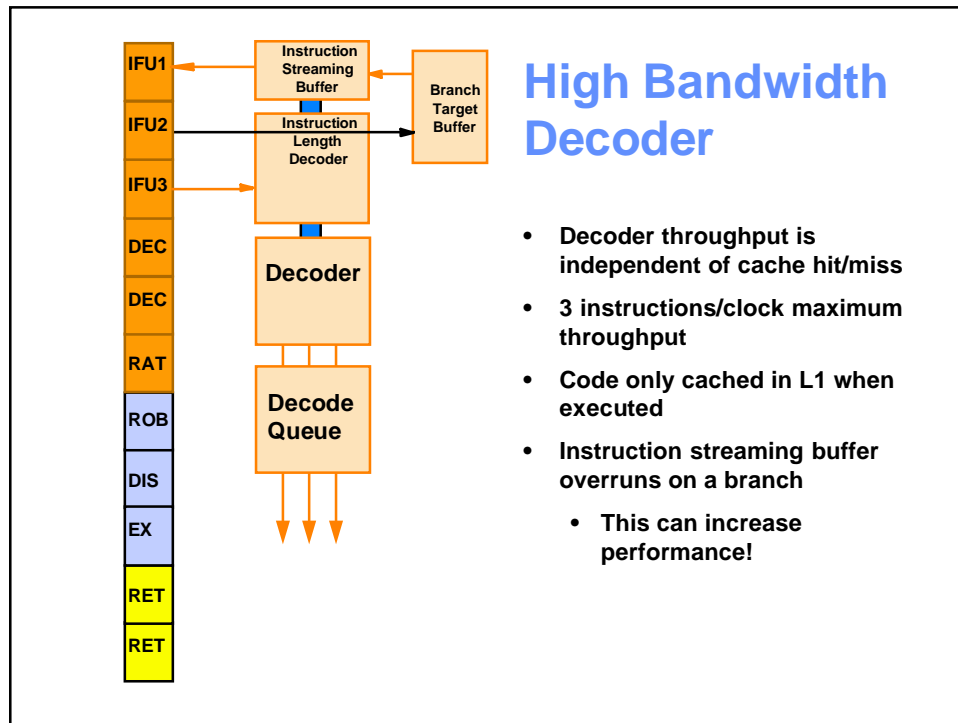
Register renaming is a key element in the P6 to allowing the out-of-order engine to work. The normal 8 general purpose registers of the Intel Architecture would become exhausted very quickly without the ability to transparently reference a larger number. The mechanism is simplistic to describe. When software references the EAX register, the internal machine references an aliased register number. Each future **read** of this register references the same register number, i.e. a true dependency. Each future write of this register references a new register number, transparently a different physical register.

The example above shows a false dependency on the EAX register within the 4 instructions. P6 would reference completely different registers for the first two instructions in comparison with the last two instructions.

The P6 has 40 physical registers apart from the 8 real registers.



In the next section, we will review the P6 pipeline, focussing on the in-order front end which consists of the initial 6 stages of the P6 pipe.



The first three stages of the pipeline are the instruction fetch unit. The first stage performs a read of the Icache. The data read, a 16 byte half cache line buffer full of code, is passed on to the second stage - the Instruction Length Decoder. This stage marks the beginning and end of each instruction and passes data on to two places:

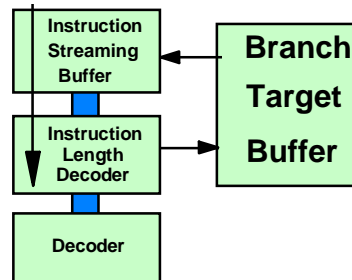
1. The first destination is the BTB, where a target address lookup is performed. If a valid target is found a new IFU address is presented to the first stage and the new code fetched.
2. The second destination is the third stage of the IFU. This is the instruction rotation stage where instructions are rotated to align exactly with their respective decoder units.

The performance of the IFU and decoder combination is independent of where the code is fetched from, cache or memory. That is, no pre-decoded information is stored with the instructions as in the Pentium® processor architecture.

The cache line overrun caused by the BTB prediction being one clock cycle down the pipeline actually can improve performance.

Large Branch Target Buffer

- 512 Branch to/from entries cached
 - Branch targets only cached when seen taken
- 2 level adaptive algorithm
- Static branch predictor
- Return Stack Buffer
 - Reduces misprediction for RET instructions



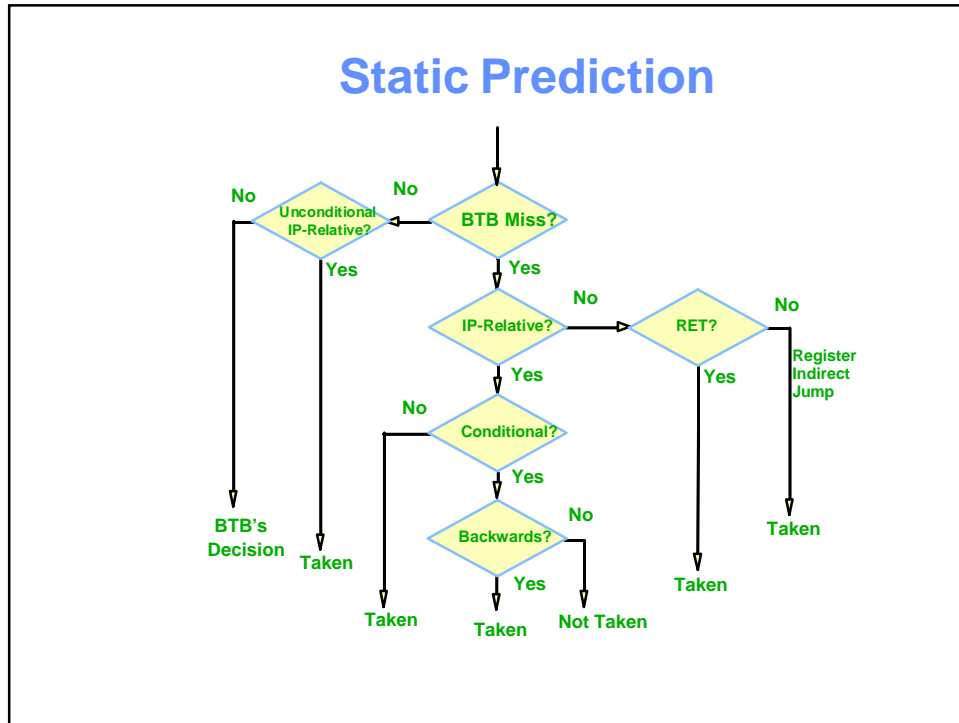
The BTB will only make a prediction on branches it has seen taken previously.

The BTB is twice the depth and width of the Pentium® processor's implementation. It can present the front end of the IFU with up to 4 new target addresses per 16 byte line. (Note, this is different than the normal 32byte cache line size). The Pentium processor implementation was capable of only delivering a single prediction.

The BTB also implements a Return Stack Buffer (RSB). The RSB will turn the almost certain branch misprediction caused by the return statement at the end of a sub-routine into an almost certain correct prediction. This works very much like the processor's internal stack mechanism.

The BTB does not implement the static branch prediction mechanism but does work in conjunction with it. In the absence of a BTB prediction on a decoded branch, a static prediction will be made. The BTB is then updated.

Static Prediction



The static prediction algorithm is depicted in the flow diagram above. It is best exemplified by following the decision tree down the center. That is, if there has been **no prediction** from the BTB, **and** the branch is **relative** to the current EIP, **and** the branch is **conditional**, **and** the sense of the branch is **backwards**, then it will be predicted to be taken by the static algorithm. The BTB will be updated so that the 5-6 clocks expended this time will be reduced to one the next time this branch is seen.

Can we help the P6 Predict Branches?

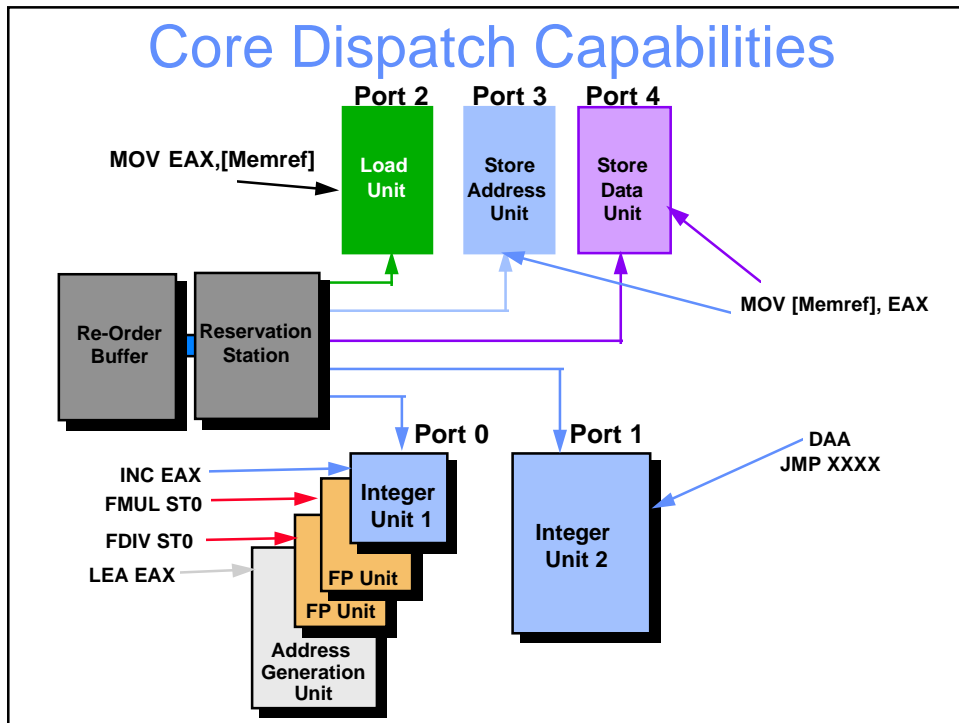
YES!

- **Get rid of branches when possible**
- **Make those remaining more predictable**
- **Observe static prediction algorithm**
- **Profile driven code reorganization will improve performance**
- **New CMOV instruction can reduce branch frequency for higher performance (loses backwards compatibility)**

P6 execution performance benefits considerably from a high frequency of correct branch predictions. Several techniques can help improve the branch prediction hit rate.

Compilers that understand and implement the static prediction algorithm, Proton V2.0, will see an improved level of performance. This improvement will vary from application to application.

Heuristic profile feedback is starting to be used at the high end of the application marketplace. The feedback mechanism allows the compiler/linker to create a “packing” of frequently executed code into contiguous units with predictable branches forming the decision loops. Proton V2.0 has this capability.



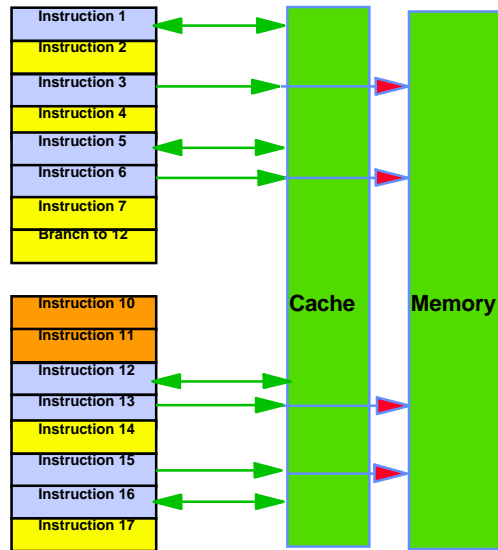
The Reservation Station (RS) has the task of dispatching ready UOPS off to the execution units. The P6 has 5 internal ports upon which we can dispatch UOPS. The P6 can dispatch and writeback 5 UOPS in the same clock cycle (simultaneously).

The execution capabilities are encompassed by:

- Port 0 - Two Address generation units
Two Floating point units
Integer execution unit
- Port 1 - Integer execution unit (higher capability)
Branch execution
- Port 2 - Load Unit
- Port 3 - Store address unit
- Port 4 - Store Data unit.

Smart Internal Caches

- 256K 4-way set associative second level cache
- Full speed, split transaction L2 data path
- Non-blocking cache structure
- Capability for 4 active outstanding cache misses
- Other loads placed in load buffer (12 deep)

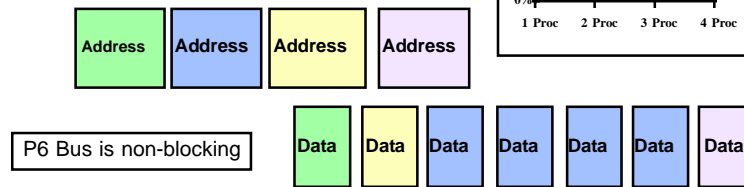
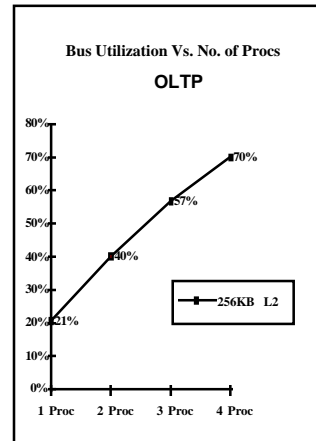


Having the ability to execute OOO as shown on the previous slide would be of little value if the engine were stalled by a “conventional” cache design.

The caches in the P6, both L1 and L2, are non-blocking. That is, they will initiate the actions necessary to return data to cache miss while they respond to subsequent cached data requests. The caches will support up to 4 outstanding misses. These misses translate into outstanding requests on the P6 bus. The bus can support up to 8 outstanding requests at any moment in time. The cache will also “squash” (suspend) subsequent requests for the same missed cache line. Squashed requests are not counted in the number of outstanding requests above. Once the engine has executed beyond the 4 outstanding requests, subsequent load requests are placed in the (12 deep) load buffer.

Transaction-Oriented P6 Bus

- Demultiplexed
- 64-bit data
- 36-bit physical address
- O-O-O
- 8 Outstanding bus requests allowed



The P6's non-blocking core capabilities are propagated out onto the system bus. The transaction based bus removes the execution suspension characteristics associated with a coupled bus as in the Intel486™ Processor and the Pentium® Processor.

The bus is also capable of reordering transactions, via the DEFER response, to support long latency accesses (i.e. going through some slow I/O or intercluster controller).

The bus has been simulated at being 21% loaded in an on line transaction processing (OLTP) environment in which it supports > 400TPS. The bus obviously supports multiprocessing and in the case where there are 4 processors executing in a OLTP environment, the system performance scaling factor is ~3.1 on transaction processing workloads.

Memory Effects of OOO

- What does O-O-O do when it hits memory?
 - Intel Processors go to memory quite a lot
 - If we allow a true weakly ordered system, would lose old binary compatibility!
 - Strong ordering leaves a lot of performance on the table (>25%)
- What options can we allow?
 - Loads can pass loads (helps a lot)
 - Loads can carefully pass stores (helps a lot)
 - Stores **CANNOT** pass stores or else it is 'FIRE, READY, AIM'

```
MOV EAX, Bullet
MOV [AMMO], EAX
MOV EAX, Target
MOV [SIGHT], EAX
MOV EAX, "Fire"
MOV [TRIGGER], EAX
```

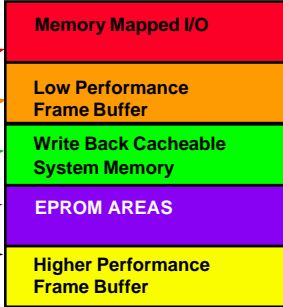
How do we control where we can do what?

The P6 has implemented an Out-Of-Order engine. What would happen if we allowed that OOO execution flow to hit memory, which, because of the relatively small register set, Intel architecture processors do quite often? We would lose compatibility with existing coded algorithm, drivers, operating systems and applications! Alternatively, if we forced strict ordering we would leave quite a lot of performance on the table.

We allow the re-ordering of loads (reads) within loads. We allow loads to be executed ahead of stores (writes). This is done carefully such that stores to the same address as a previously executed load are snooped and the data from the store buffer is forwarded to the previously loaded register (store buffer forwarding).

However, stores are **NOT** allowed to pass stores. This restriction only costs us about 3-5% in performance. The code example above shows 6 instruction with 3 stores. In this example if we were to re-order stores we would have a Fire, Ready, Aim possibility -- which we cannot allow.

Memory Type Range Registers

- MTRR's are used to "color" memory regions and define what can be done where
 - Memory colors are:
 - UNCACHEABLE
 - WRITE THROUGH
 - WRITEBACK
 - WRITE PROTECT
 - UNCACHEABLE, SPECULATABLE, WRITE COMBINING
 - 8 Variable Ranges above 1M
- 

Software takes over configuration of the hardware

The mechanism which we use to control the way the P6 interacts with memory is called the Memory Type Range Registers. These are the registers that were typically contained within chip sets. These registers have been moved into the processor for performance reasons.

The memory granularity is:

64KByte 00000 - 7FFFFH

16KByte 80000 - BFFFFH

4KByte C0000 - FFFFFH

1MByte 100000 - FFFFFFFFH in 8 separate regions

The USWC memory type is a new memory type brought in by the P6.

A note for those of you that have integrated frame buffer capable video controllers in motherboard designs: You will get great video performance from the P6. You will get even better video performance if the BIOS sets the frame buffer memory range to USWC memory type.