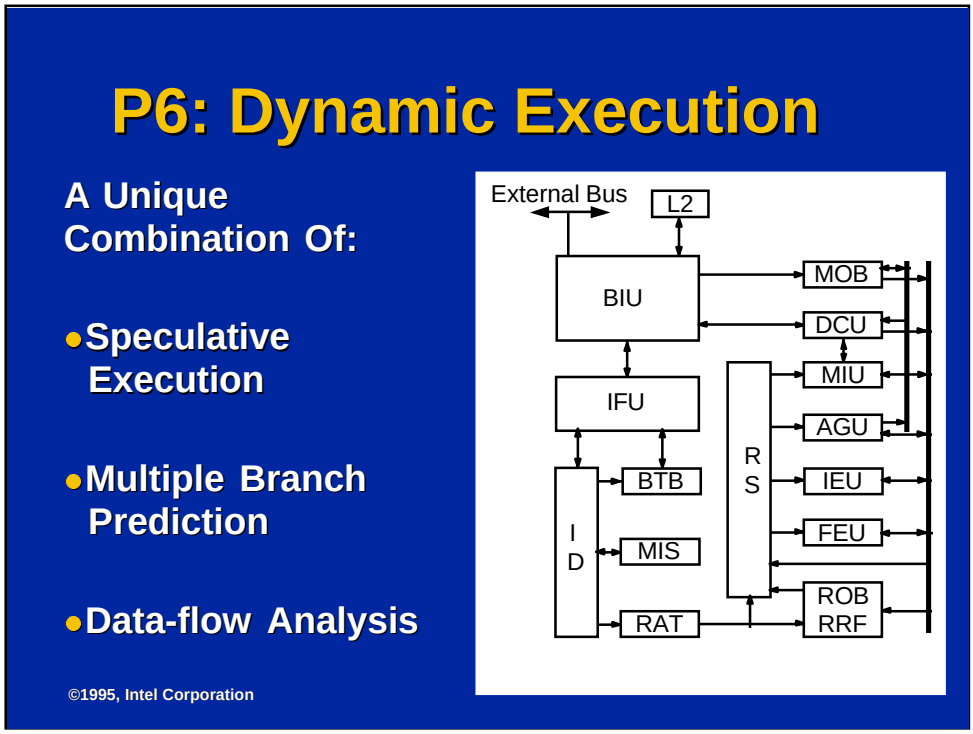
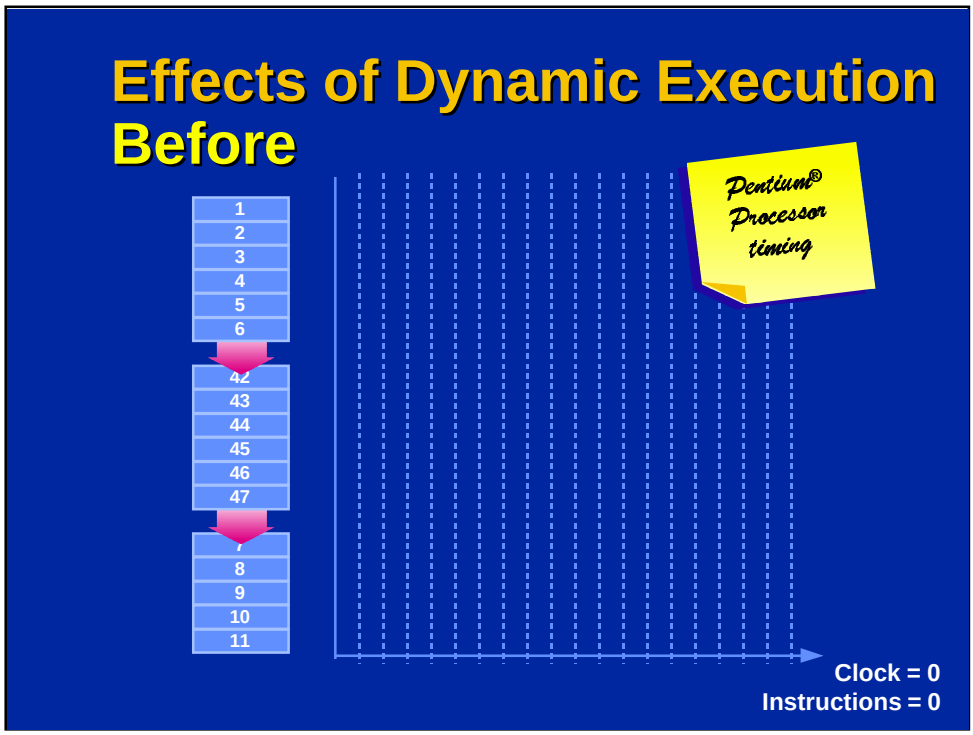


The key architectural feature of the P6 processor is Dynamic Execution -- a unique combination of speculative execution, multiple branch prediction, and data-flow analysis.



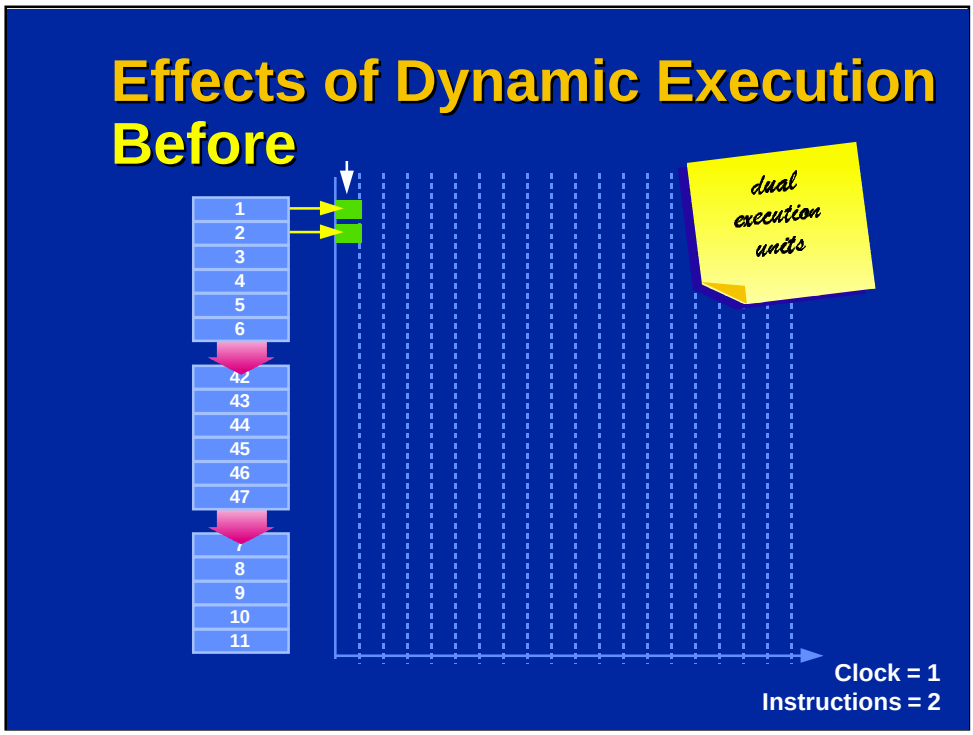
To show the effects of Dynamic Execution, we show a BEFORE and AFTER scenario.

The Pentium® processor is an excellent example of a high performance, superscalar (level 2) CPU, so I chose it as a basis for comparison (the BEFORE).

The diagram lists a set of instructions on the verticle axis -- note they start at #1, there is a subroutine call at #6 to #42, this returns to #7. What the instructions actually are is not important to this explanation.

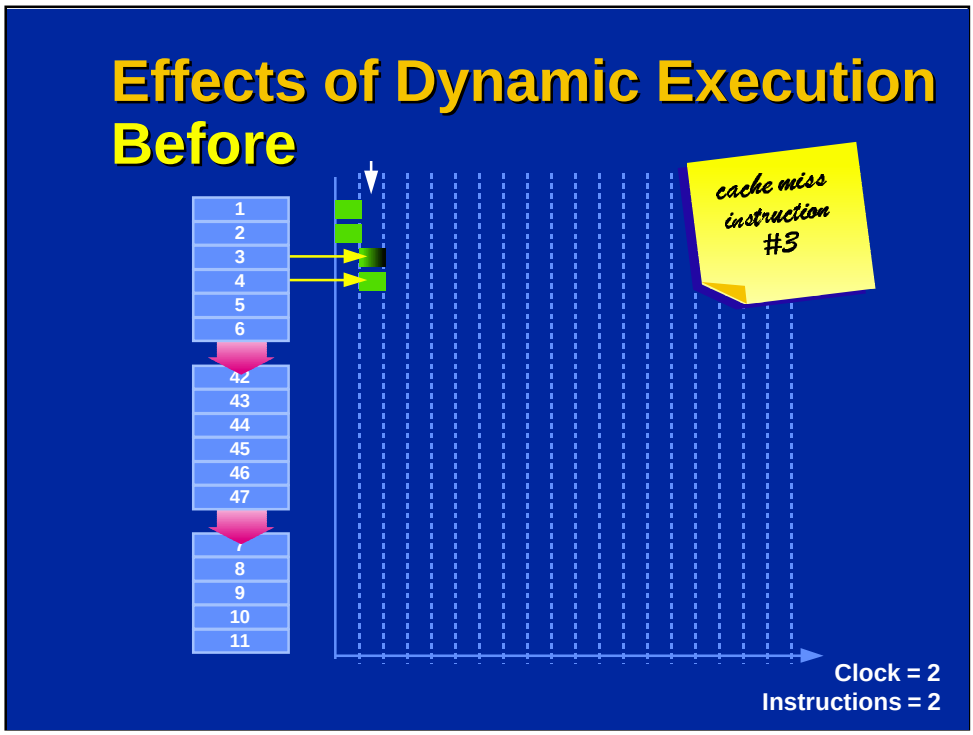
The horizontal axis is time, the vertical dotted lines are CPU clocks.

There is a tally counter in the bottom right corner that keps track of the number of clocks we have used and the number of instructions completed. (Completed is important, starting an instruction is interesting but this needs to complete to be counted.)



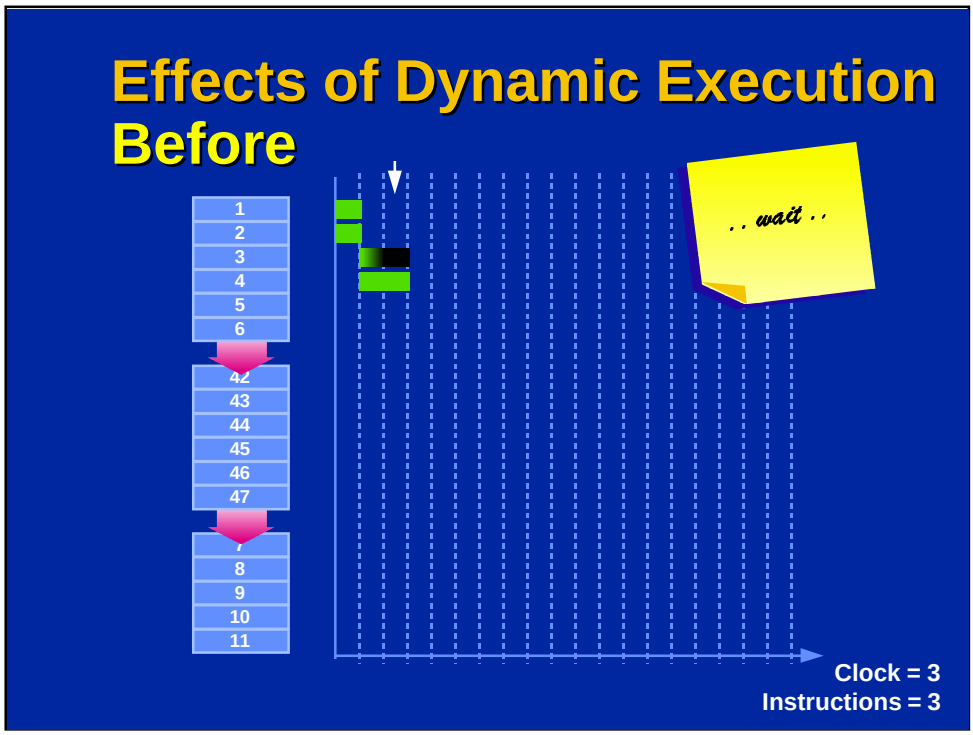
The first clock:

The Pentium® processor started and completed TWO instructions in this clock. Remember the Pentium processor has two execution pipes, U and V, and both of these instructions happened to be single cycle.

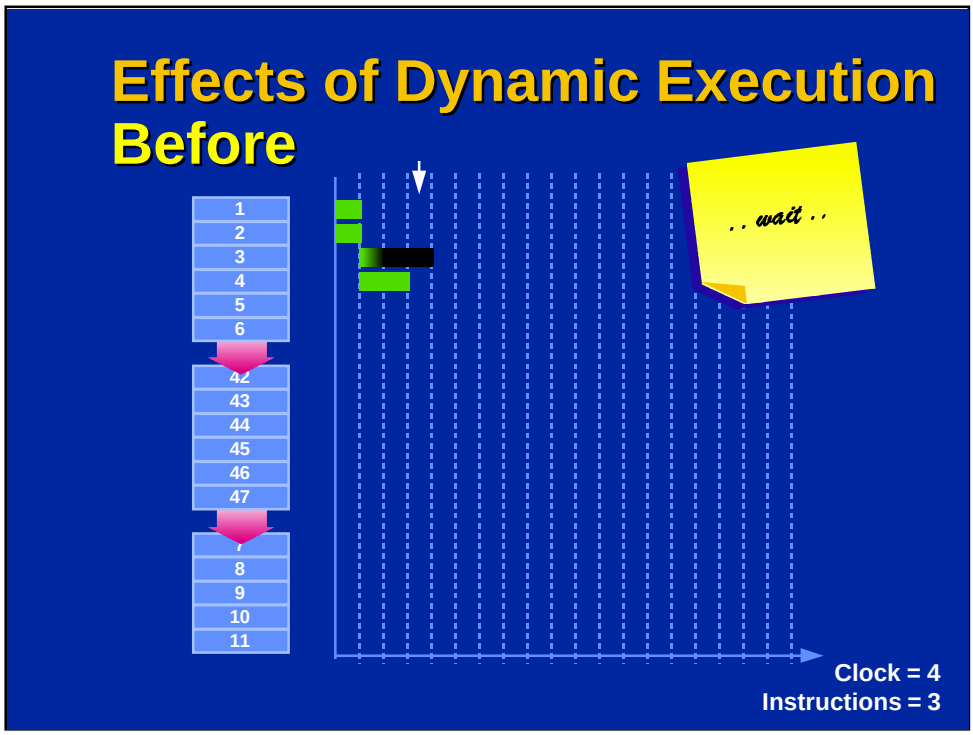


On Clock 2, the Pentium® processor starts the next 2 instructions

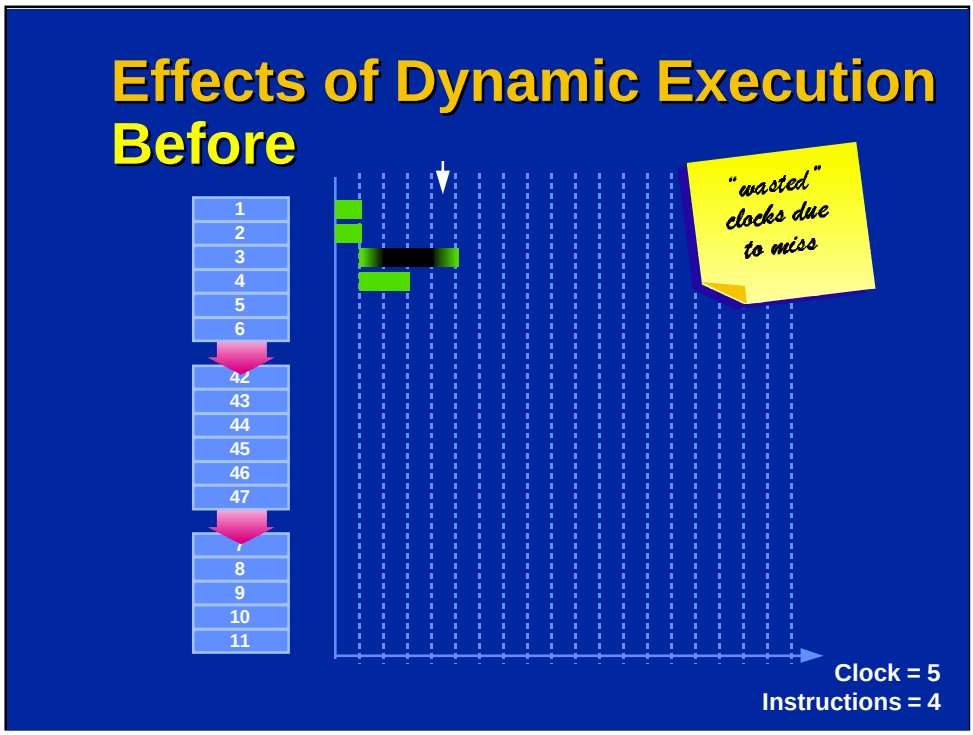
Note that #3 was a load of data and that data was not in the cache -- that is, we have a cache miss. (The instruction is colored black via a gradient shading to show this.) We will have to go and get this data from external memory (and it's going to take 4 clocks).



On Clock 3, instruction 4 completed but we are still waiting for the cache miss on #3.



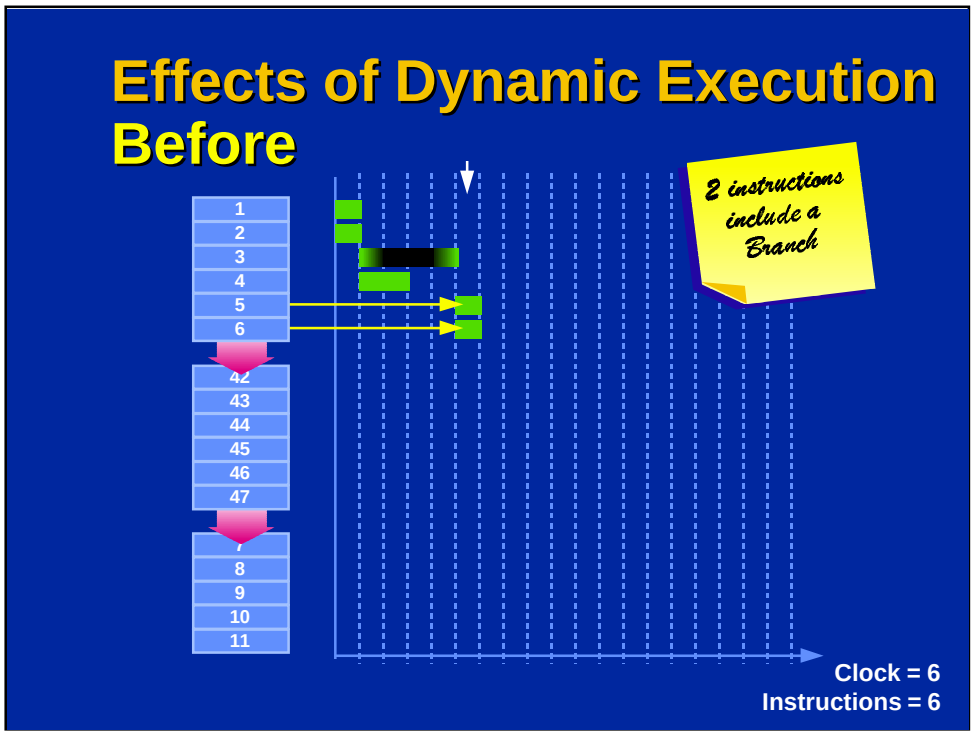
On Clock 4 we're still waiting.



Data is returned from memory and #3 completes.

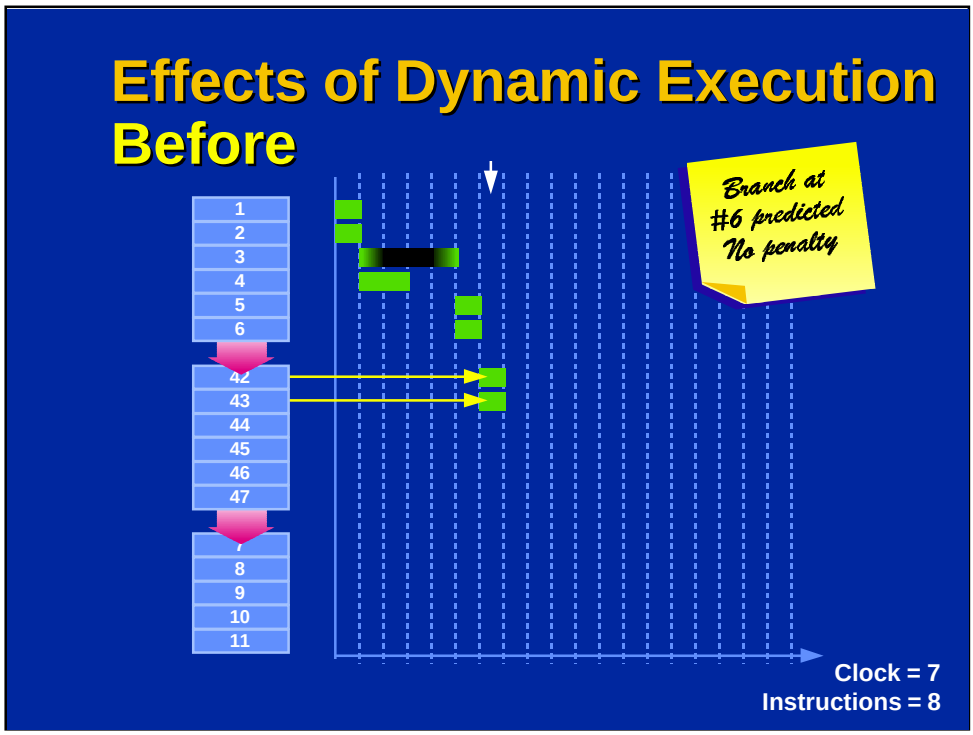
I classify the “black” area as wasted time since no forward progress was made with the program -- the CPU was STALLED waiting for main memory, or a Level 2 cache, to respond. If the data had been resident in the L1 cache, then this instruction would have completed in 1 clock.

There are, of course, technical reasons why the L1 cache cannot be made MUCH bigger to reduce this miss rate. Increasing the L1 size also has a dramatic effect on die size which, in turn, impacts the cost of the component.



In this clock, the Pentium® processor started, and completed, two instructions.

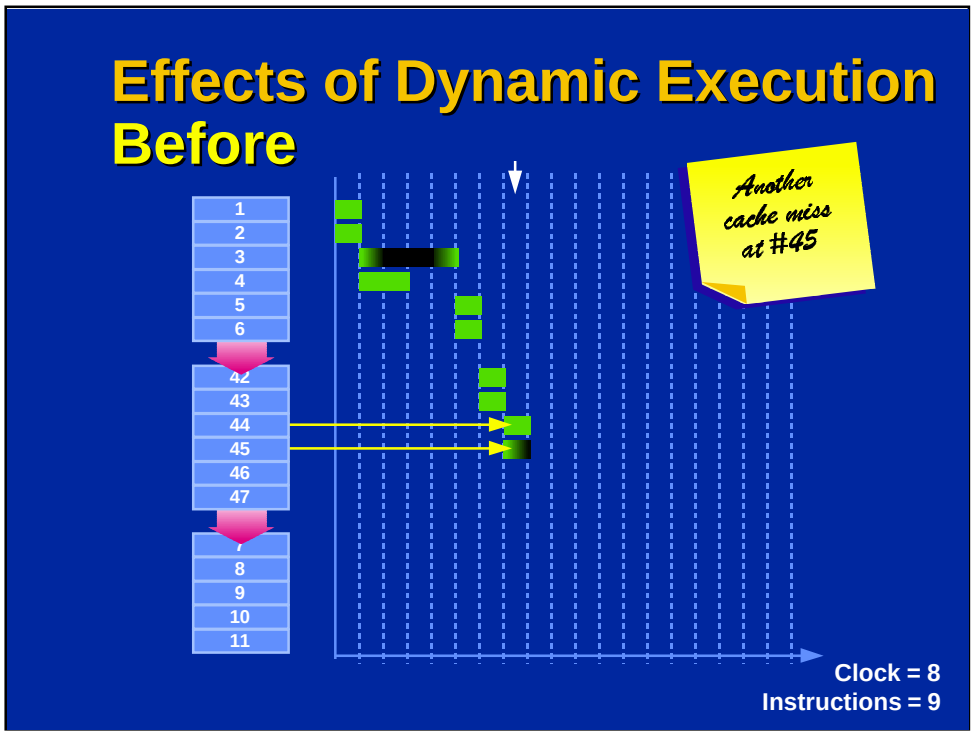
#6 is a branch that the Pentium processor predicted would transfer control to instruction 42. The processor's fetch unit is double buffered so instructions at target location (#42) have already been fetched.



The correct preduction and the prefetching allows the Pentium® processor to start two instructions at #42 and #43 in Clock 7.

There was no penalty for the transfer of control which the processor correctly predicted.

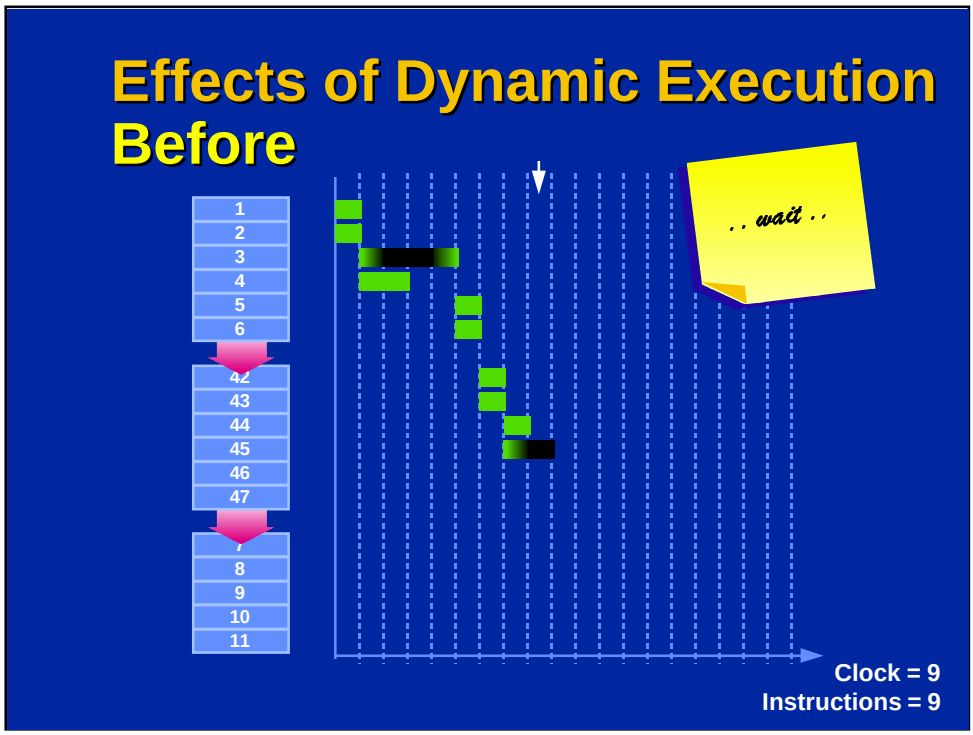
We’re at 7 clocks and have completed 8 instructions -- that’s better than 1 insruction per clock (IPC).



Inside the subroutine on Clock 8 the processor starts two instructions.

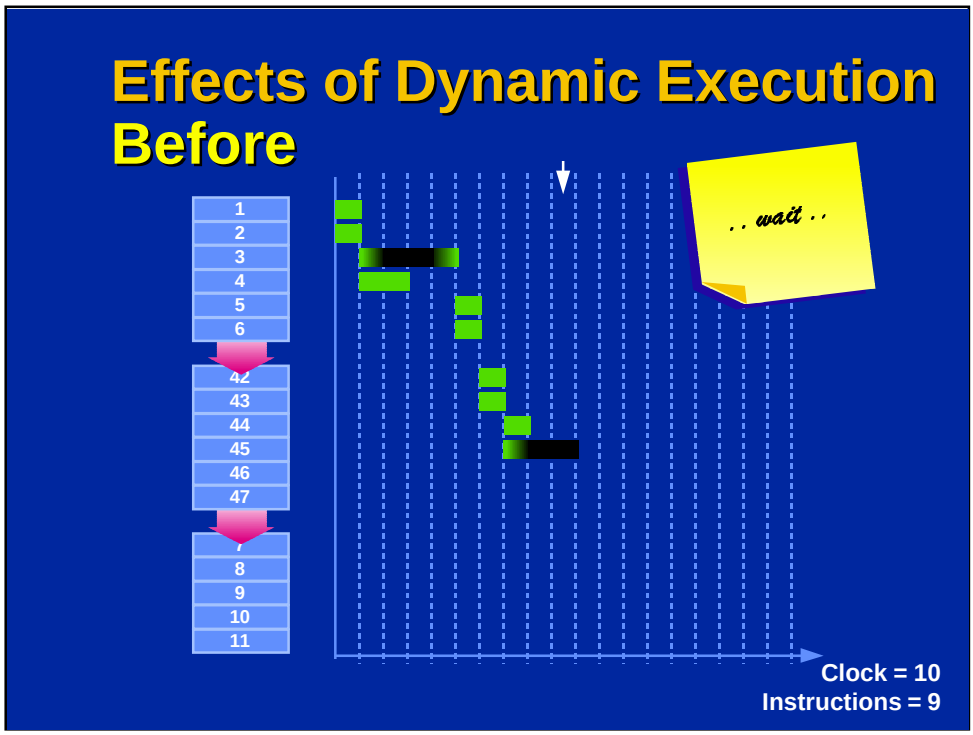
#44 is a single cycle instruction.

#45 is an L1 cache miss, the processor stalls again while the core waits and while the bus interface unit reads the data from main memory or the L2 cache.



Waiting again.

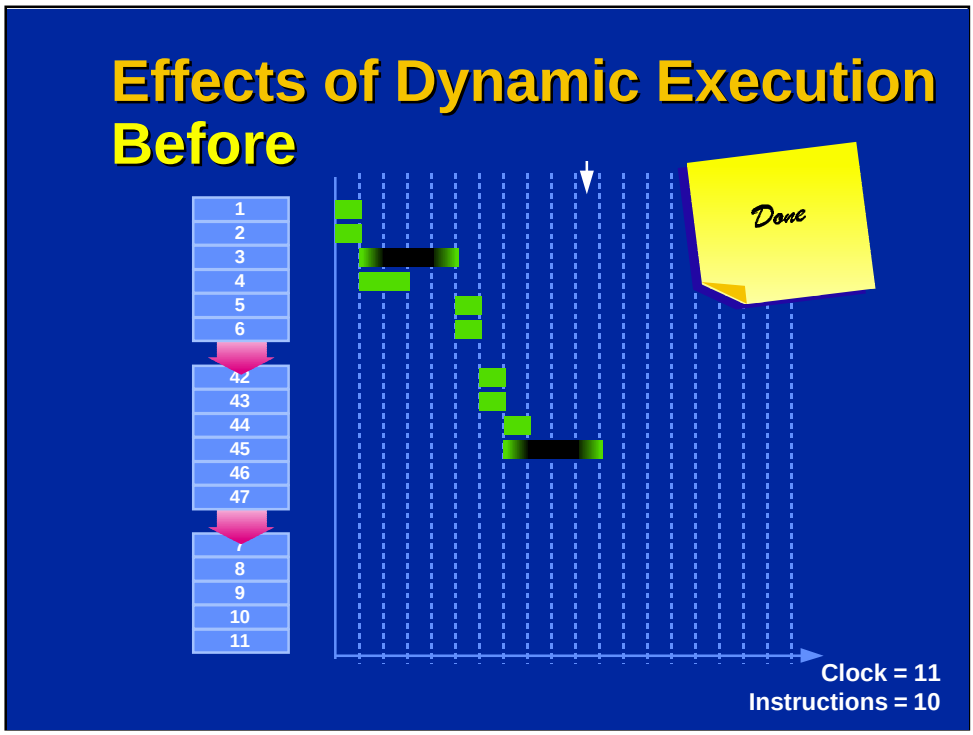
Notice that the speed of the memory subsystem is having a LARGE impact on the PP's instruction execution rate.



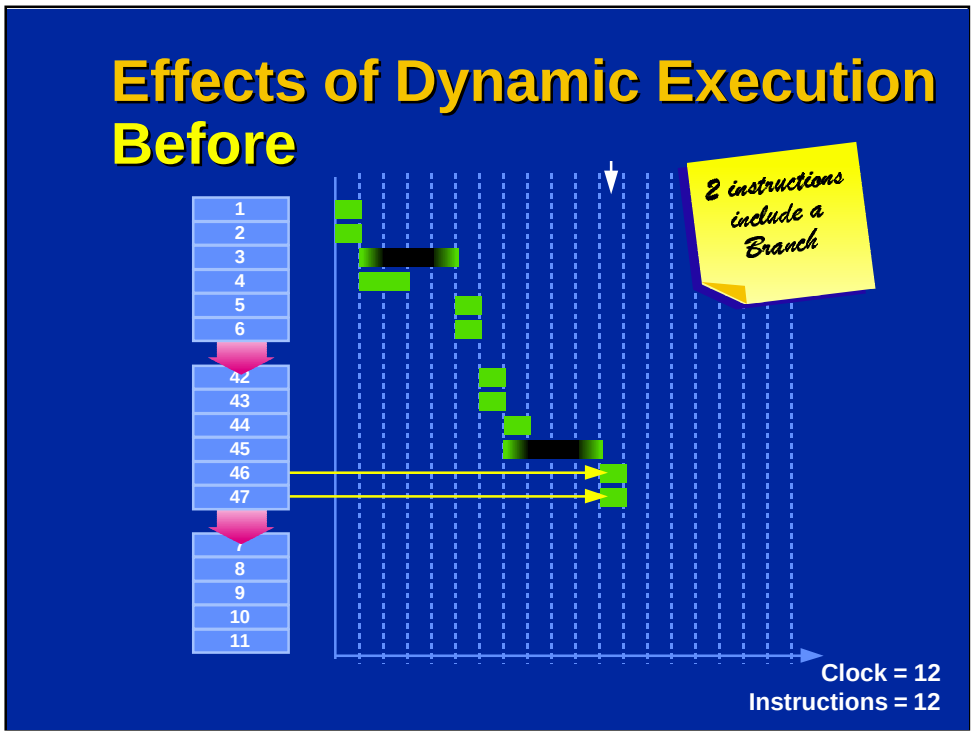
Waiting again.

The IPC has now dropped below 1.

The processor's core is stalled. It cannot do any useful work as it is waiting for data.



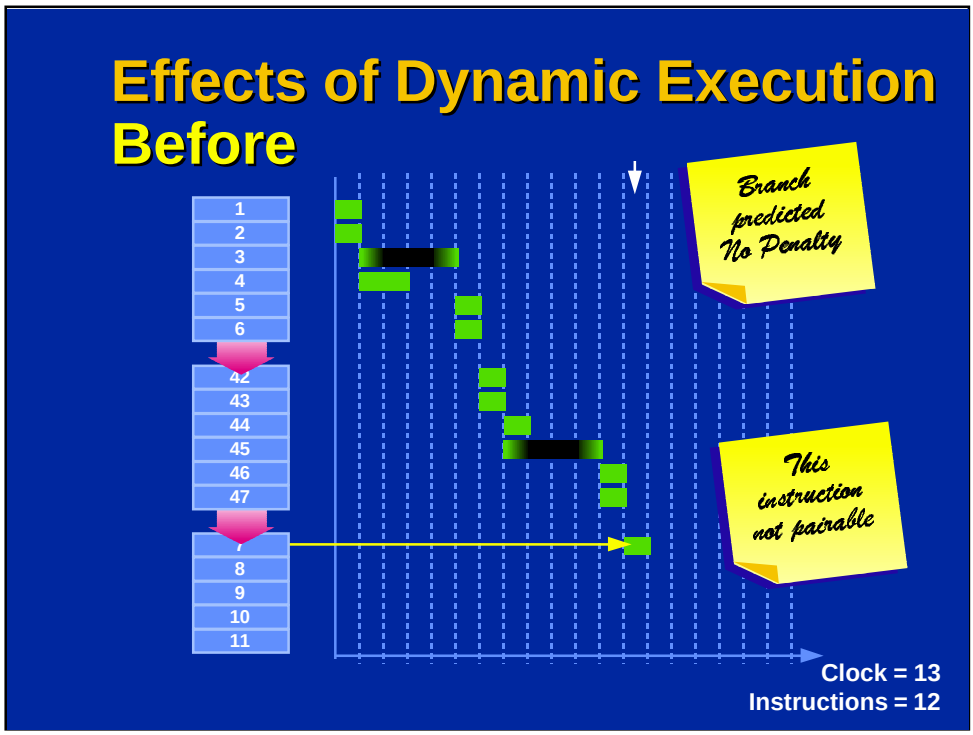
#45 completes, we can move on



Two instructions are started again.

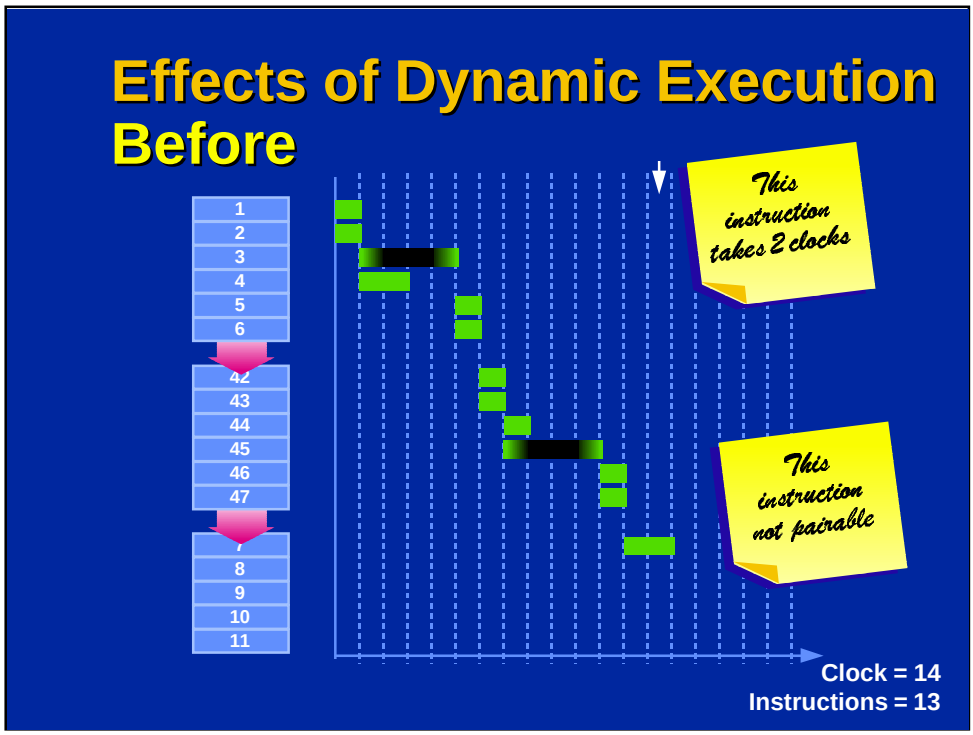
#46 is a single cycle operation.

#47 is a RETURN and the Pentium® processor correctly predicts the return address.

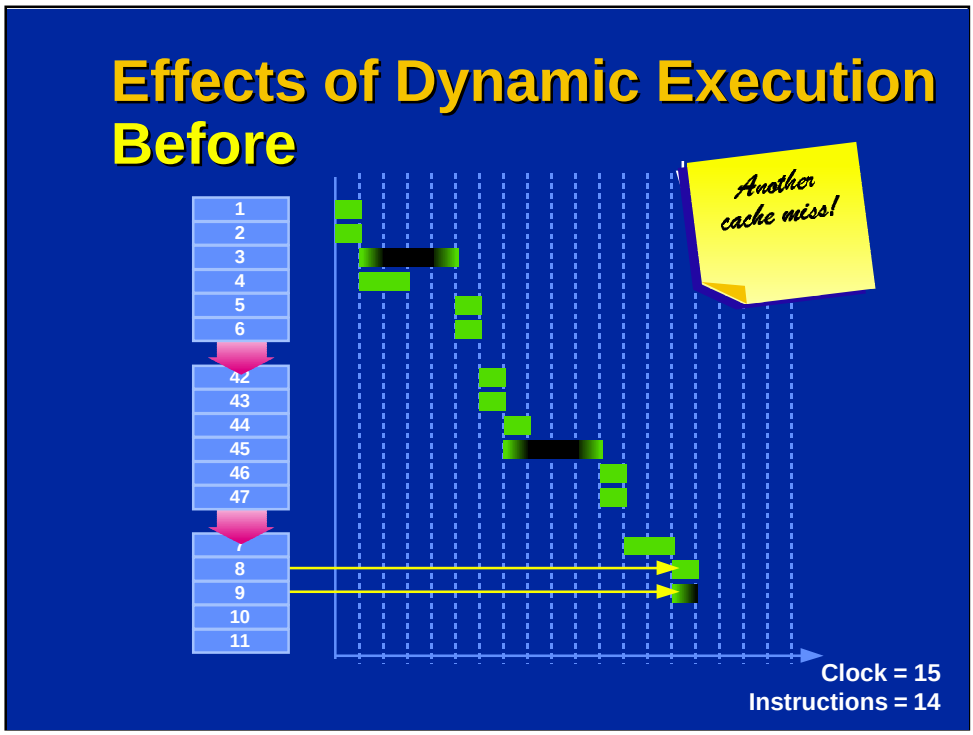


We return to the main program and #7 is started with no delay.

#7 is not pairable with #8, so the Pentium® processor only starts #7.



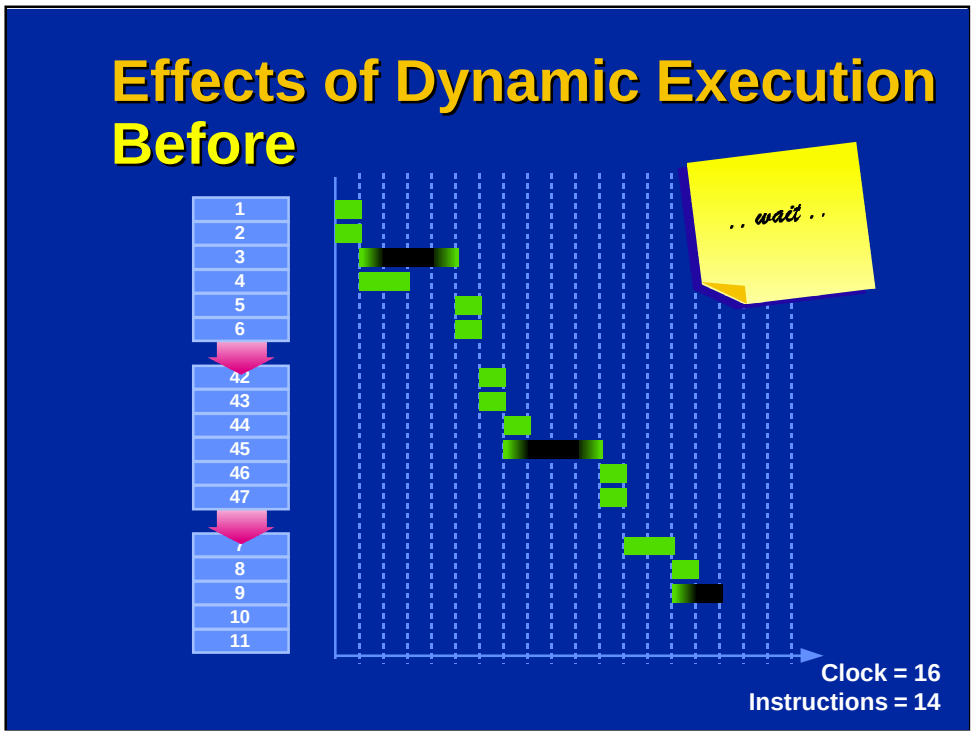
#7 is a two cycle instruction which now completes.



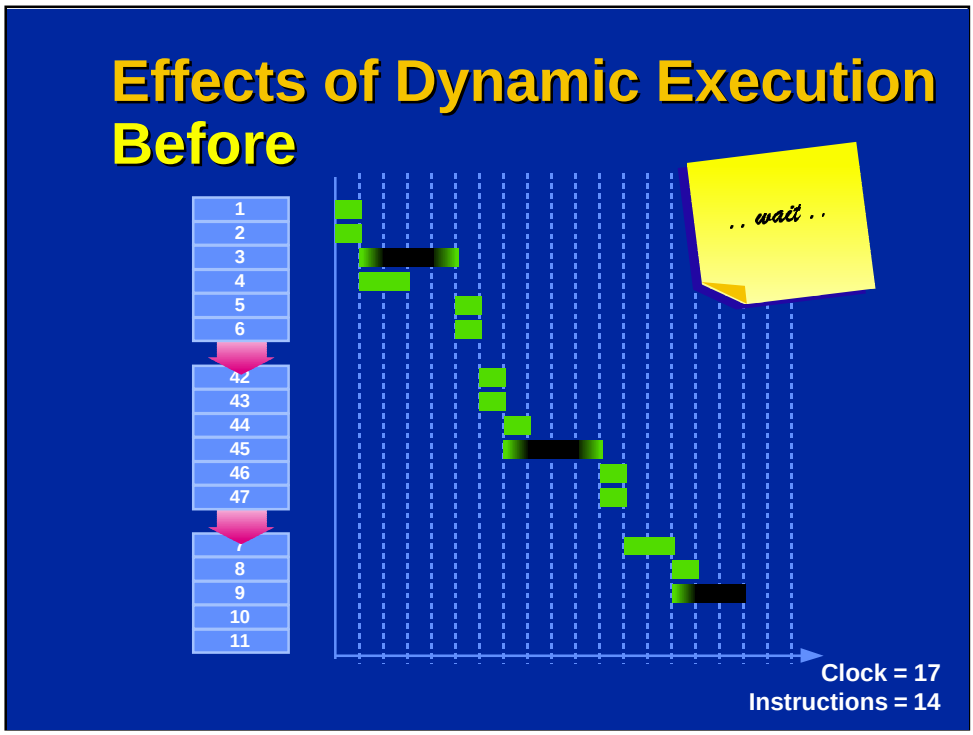
The processor starts #8 and #9 in this clock.

#8 is single cycle.

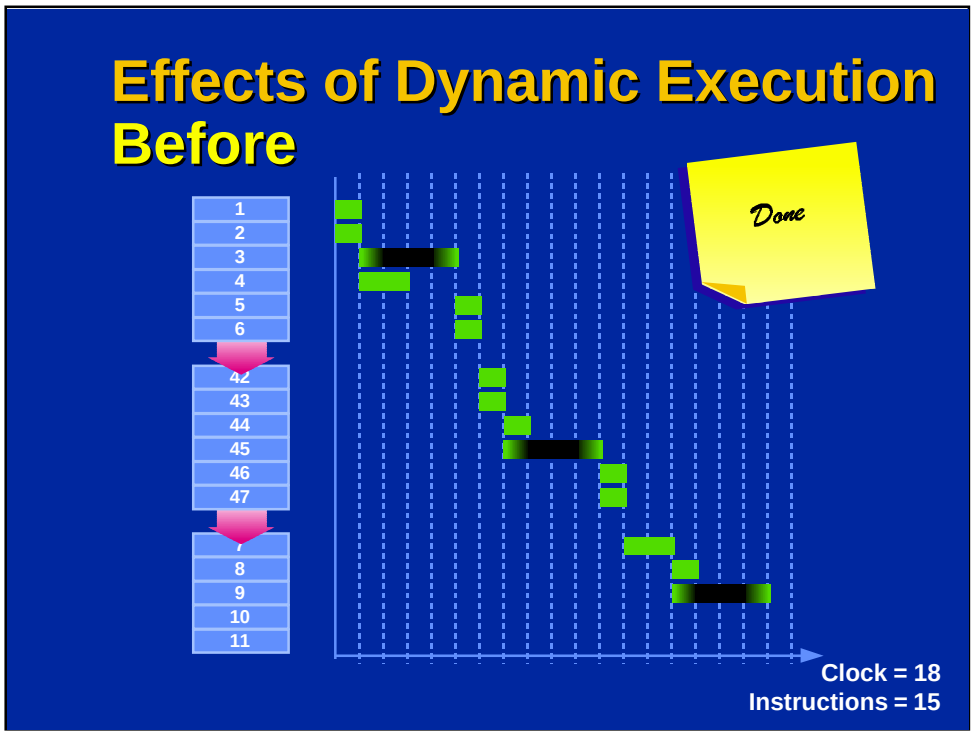
#9 is another cache miss, so we have to wait again.



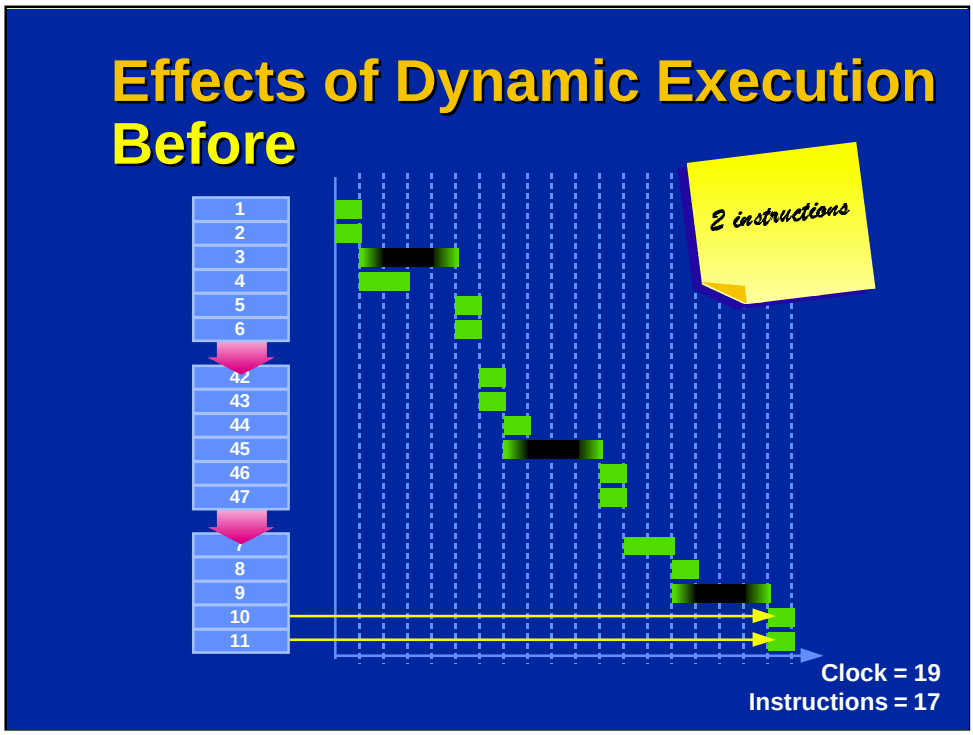
Waiting for data once more.


Effects of Dynamic Execution
Before
1
2
3
4
5
6
42
43
44
45
46
47
8
9
10
11
.. wait ..
Clock = 17
Instructions = 14

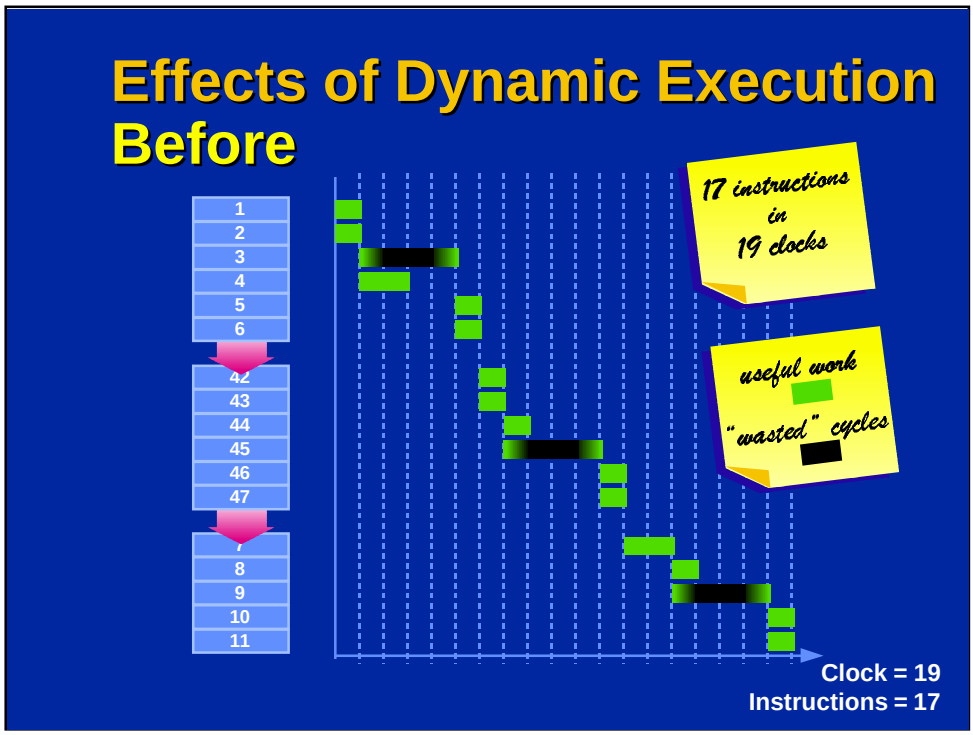
And waiting.



Done. Now let's move forward.



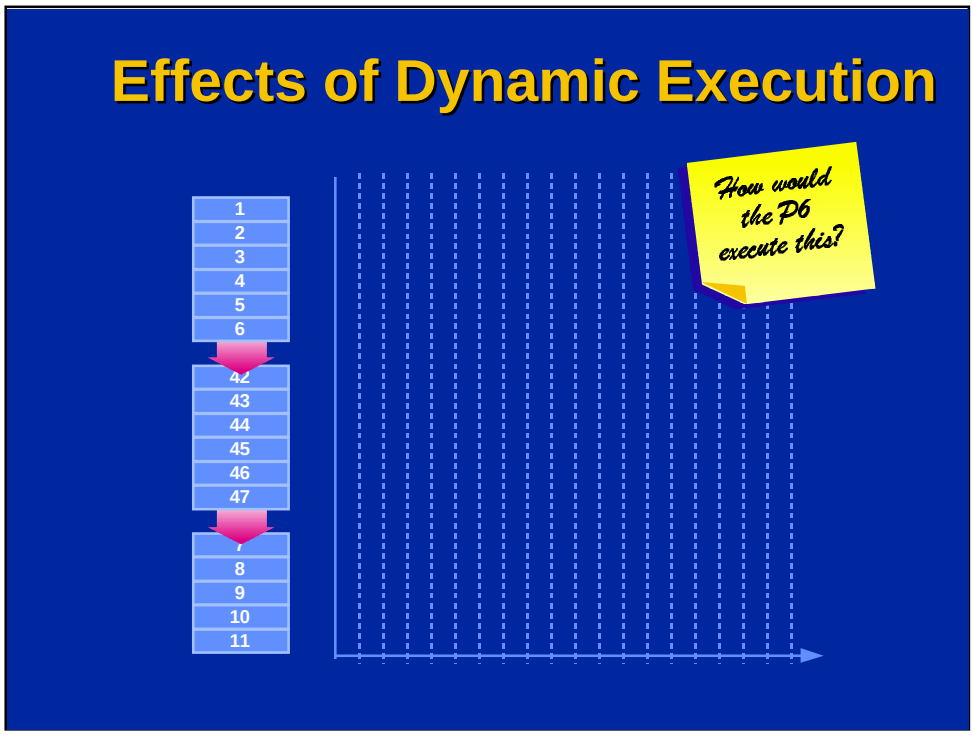
The Pentium® processor started and completed #10 & #11 to complete this sequence of 17 instructions in 19 clocks.



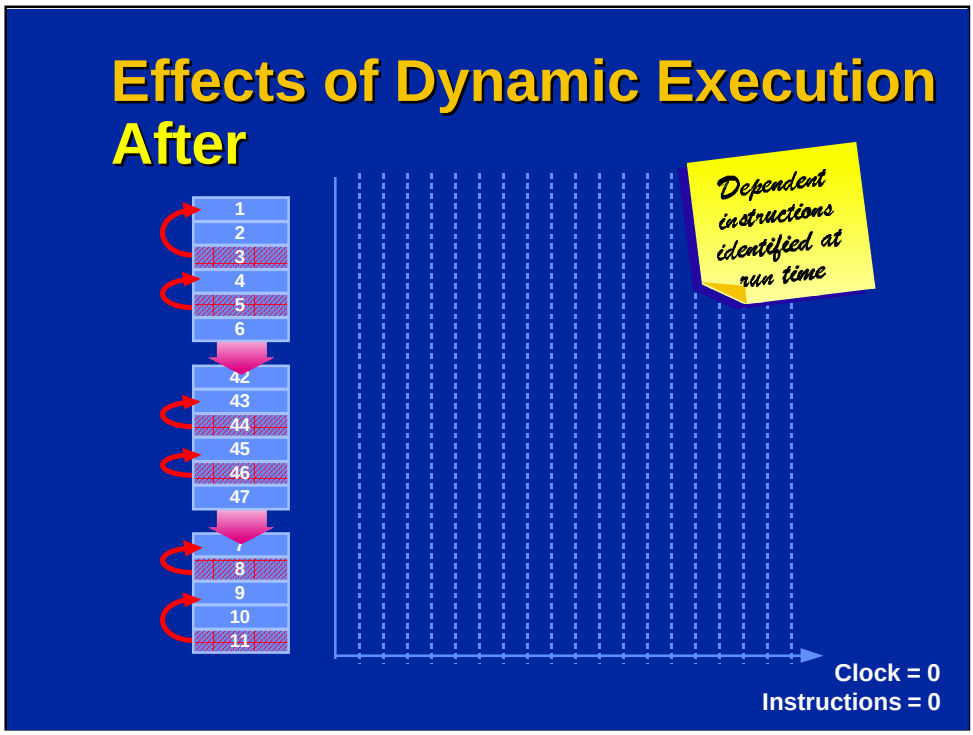
Pentium® processor summary:

Once the processor had the data, it did great work (shown in green) typically executing 2 instructions per clock (superscalar, using U & V pipes).

But a lot of time was spent waiting for data (shown in black) and this reduced our average IPC to less than 1.

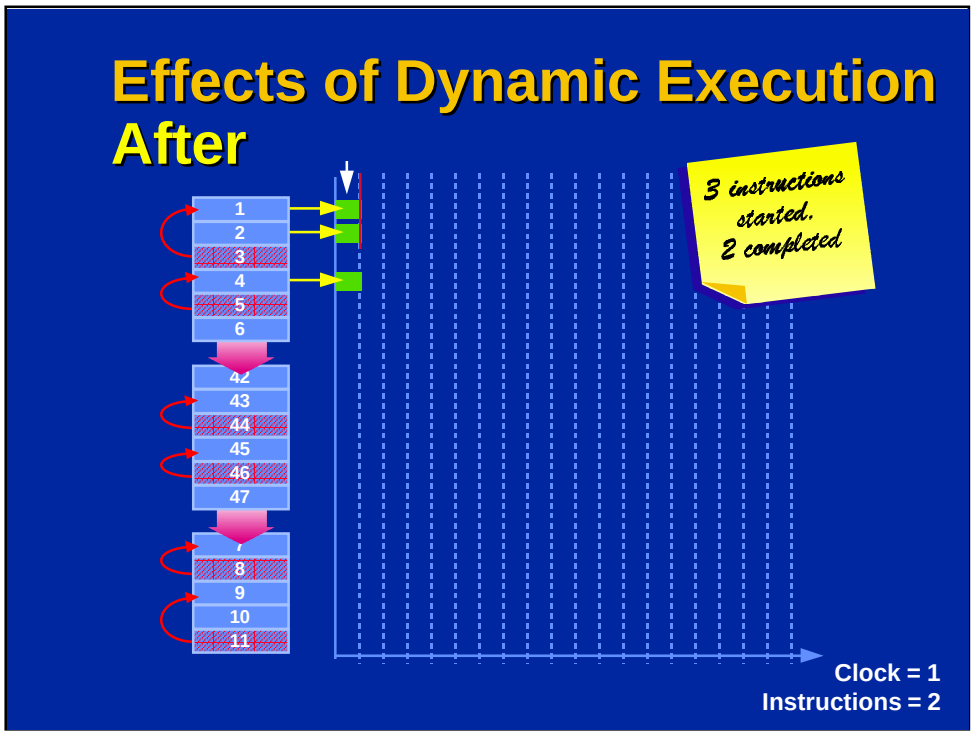


Now let’s look at how the P6 would execute this same set of instructions.



The P6 looks at all prefetched instructions and determines if there are data dependancies -- for example, #3 requires the result of #1 so #3 cannot execute until #1 has completed. Similarly #5 cannot run until #4 has completed.

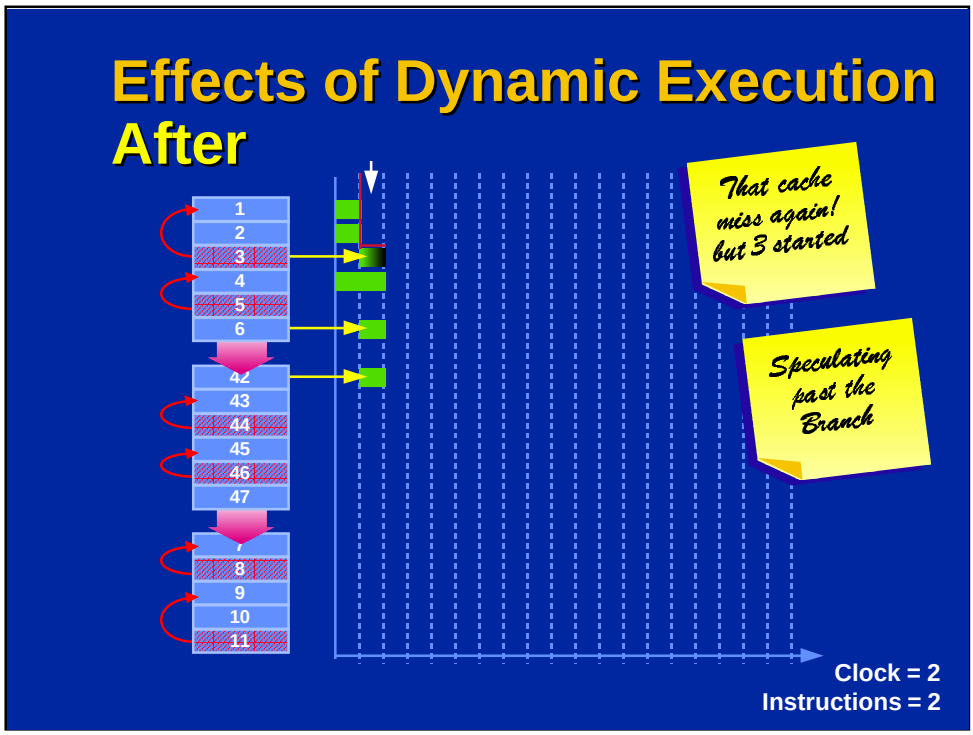
Why worry about this now? Why wasn't this a concern with the Pentium® processor? The Pentium processor executes instructions strictly in the original program order (there are some pairing issues with U & V pipes) so instruction dependancies were not a concern with Pentium processor. In contrast, we will see that the P6 is going to start many instructions and instruction dependancies will be a key concern.



On the 1st clock, the P6 starts 3 instructions -- it is superscalar level 3.

#1 and #2 are single cycle instructions and they start AND complete in this cycle.

#3 cannot start since it is dependant upon the result of #1, so the P6 also starts #4.



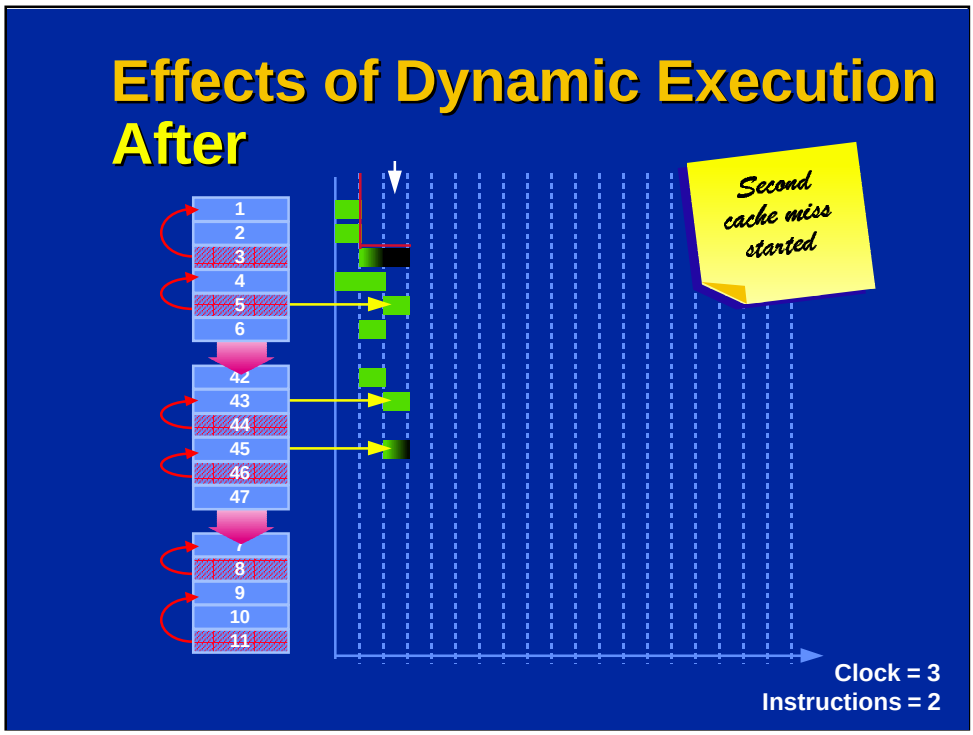
On the second clock the P6 starts 3 more instructions -- one of them is #3 which misses the cache again (as in did in the Pentium® processor scenario). We must wait for the memory access again.

Instruction 4 completes but the P6 cannot use the result yet since it must retire instructions in the same order that the program was originally written. So the P6 stores the result in temporary storage. We have speculatively executed #4.

#5 did not start since it is dependent upon the result of #4.

#6 started and completed -- this was a branch to the subroutine at #42. #42 was also started and completed.

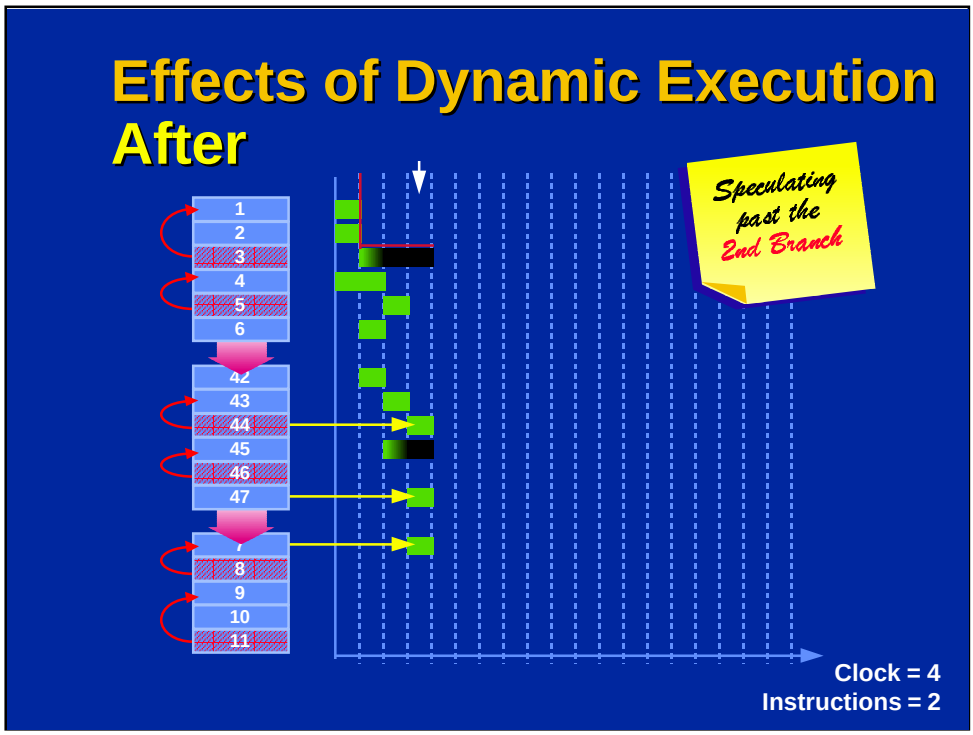
Instead of waiting for the memory access of #3, the P6 went off and did useful work by starting instructions after #3 and stores their results in temporary on-chip registers.



It's Clock 3 and we're still waiting on that first cache miss.

We start 3 more instructions in this clock too: #5, #43 and #45.

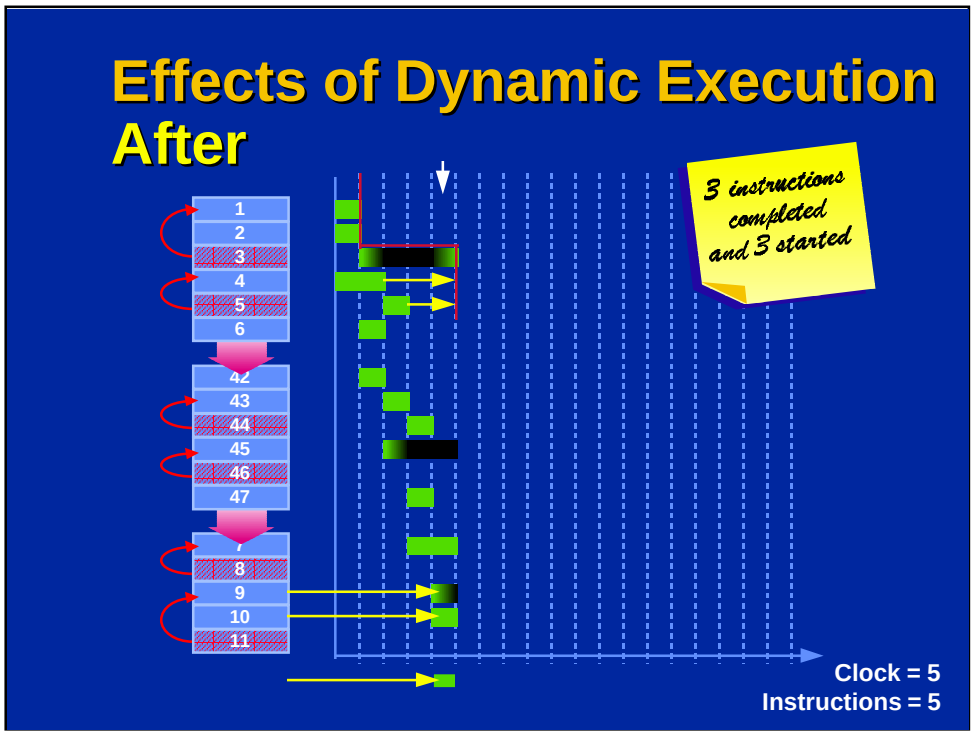
Notice that #45 is also a cache miss -- the P6 L2 cache is fully pipelined and can support four concurrent accesses. This means that we do not have to wait until the cache miss at #3 is serviced -- we can service the cache miss at #45 in parallel.



It's Clock 4 and we're still waiting on that first cache miss. We start 3 more instructions in this clock, too: #44, #47 and #7.

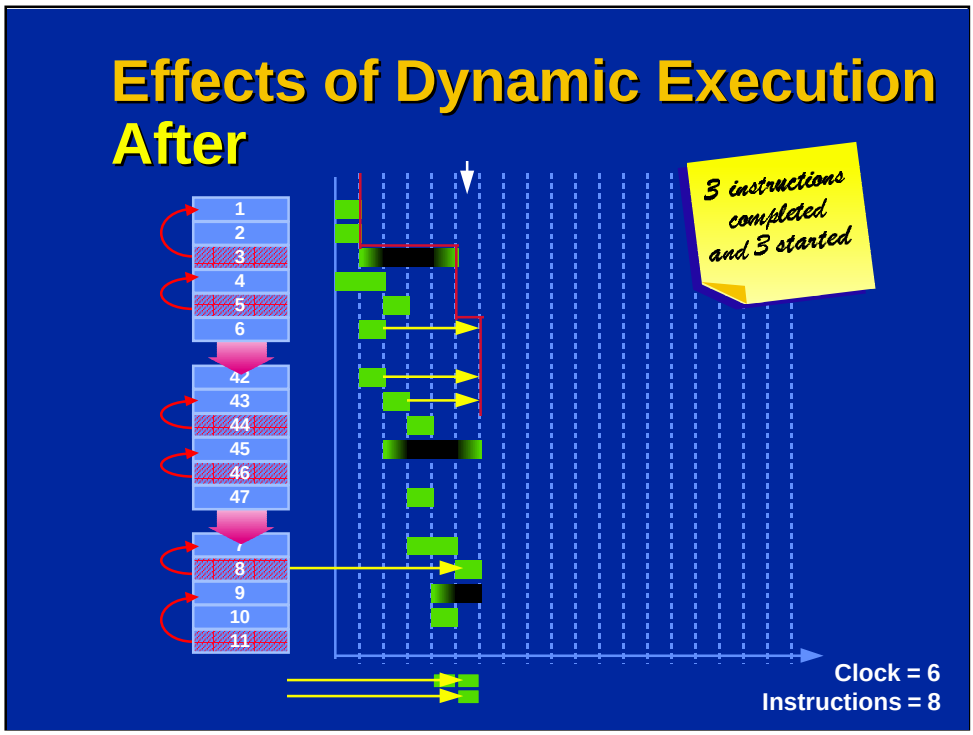
Notice that #47 is another branch (return from the subroutine). The front end of the P6 has predicted that we'll return to #7 and, indeed, we start #7 in this clock.

While the "back-end" of the P6 is waiting for #3 to retire, the "front end" has made the subroutine call and has come back! This is typical of how the P6 speculatively executes well ahead of the current program counter.

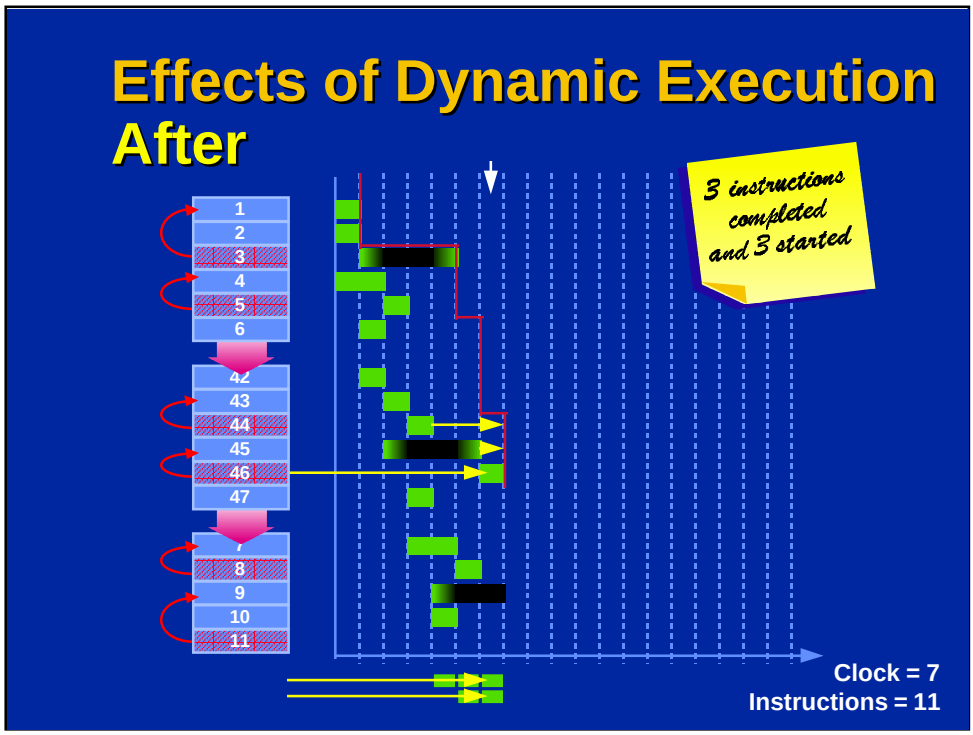


It's Clock 5 and that cache miss at #3 is finally satisfied, so the P6 retires #3 and #4 and #5. The P6 can retire 3 instructions/clock.

The P6 also starts 3 more instructions, #9, #10 and one > 11. Note that #9 is another cache miss and that, once again, we are starting the memory access early so the data will be ready when we need it later.

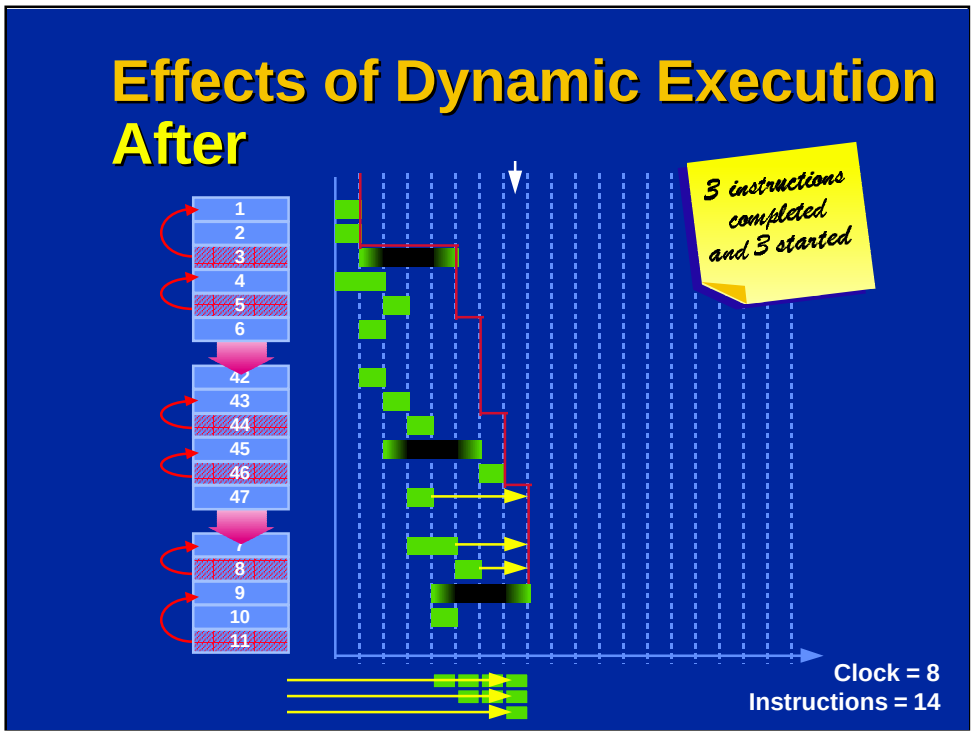


We're now at clock 6 and the P6 is "on-a-roll" -- it's starting 3 instructions per clock and completing three instructions per clock. These instructions are unrelated to each other. The P6 "front end" is finding three new instructions per clock and the "back end" is retiring three of the speculatively executed instructions per clock.

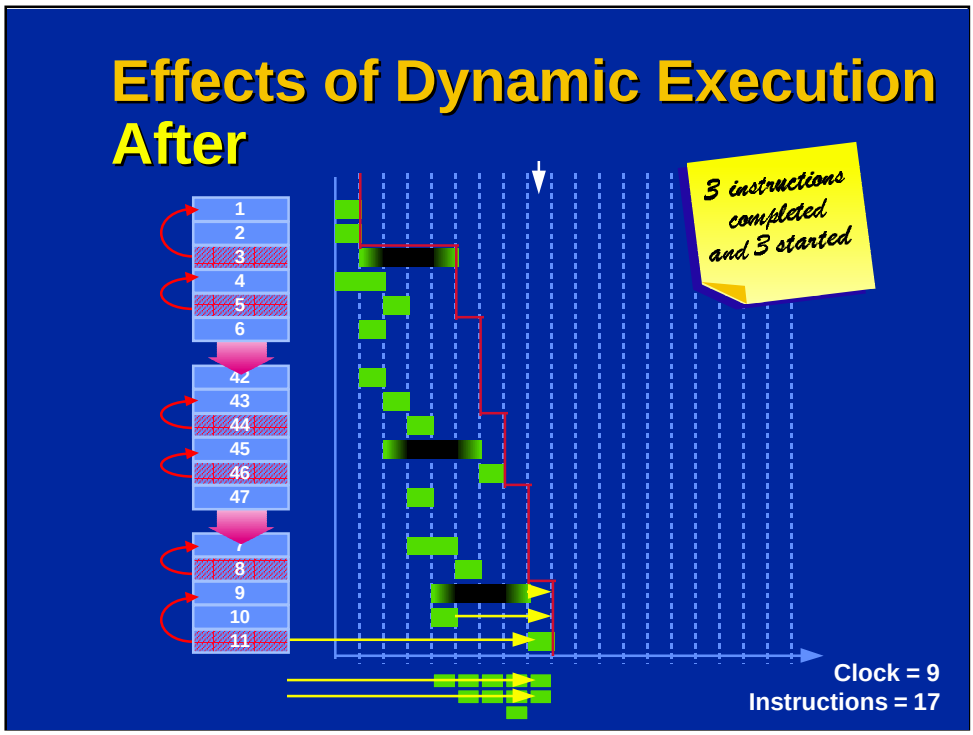


Clock 7 and, once again, we completed three instructions and retired three instructions in the same clock.

The core is observing all instruction data dependancies and is executing instructions to get maximum work done.



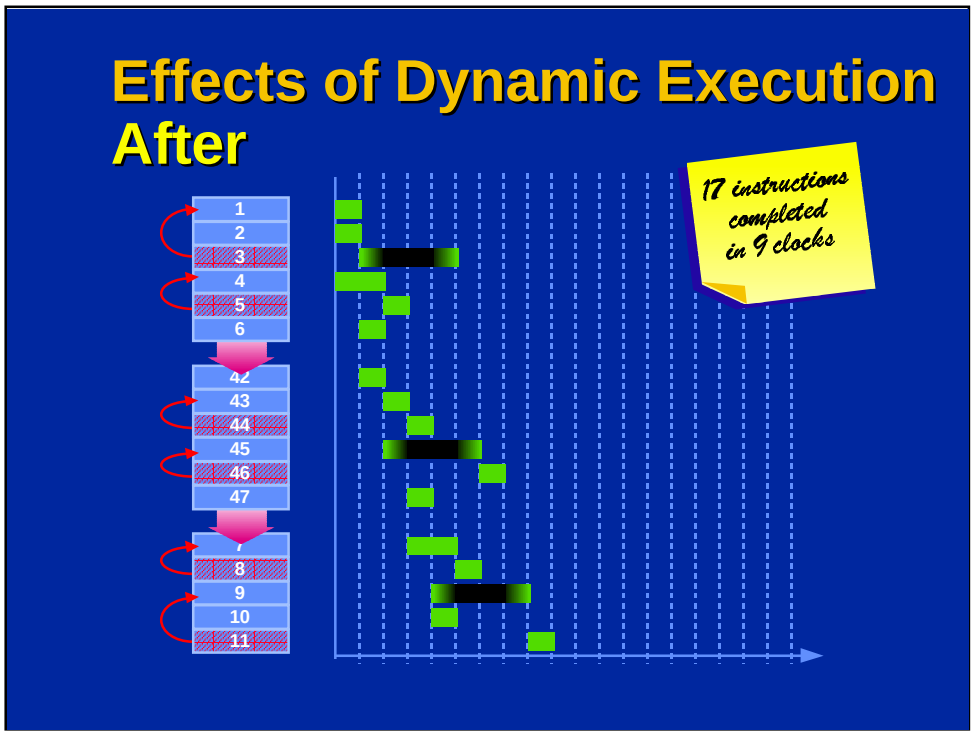
Another typical clock cycle with 3 instructions completed and 3 instructions started.



Another 3 instructions completed and 3 started. Note that #11 started and completed in this clock.

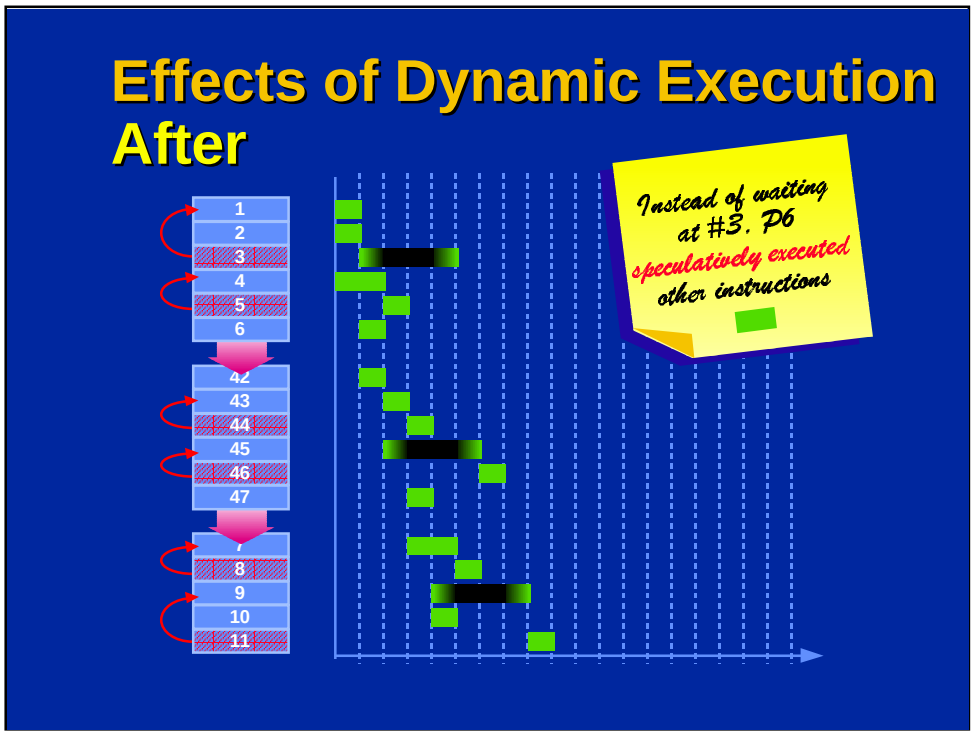
We've completed the 17 instructions (and have 10 others speculatively executed ready to retire in upcoming clocks) in only 9 clocks.

This is more than two times the instructions per clock relative to the Pentium® processor for this example.



Summary of how the 17 instructions were executed in the P6:

Note that the order that the instructions were started in is mainly a function of the data dependancies within the instruction stream. (This also depends upon availability of internal resources.) Instructions are always retired in original program order, maintaining full compatibility.



Recap of the 3 main features of Dynamic Execution:

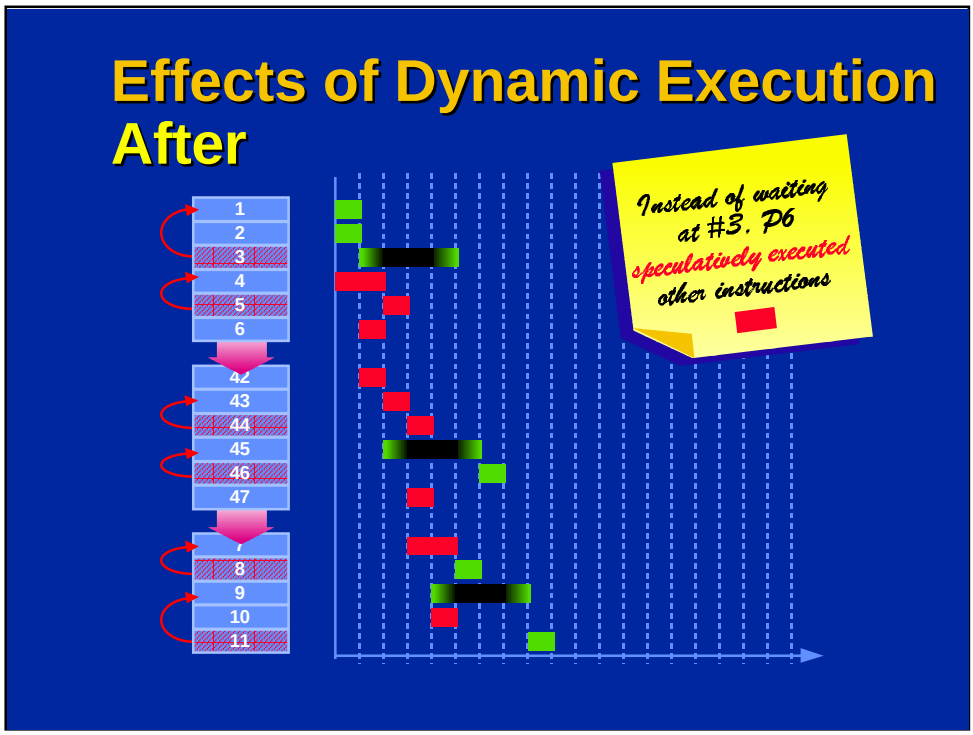
-Multiple Branch Predition

-Data Flow Analysis

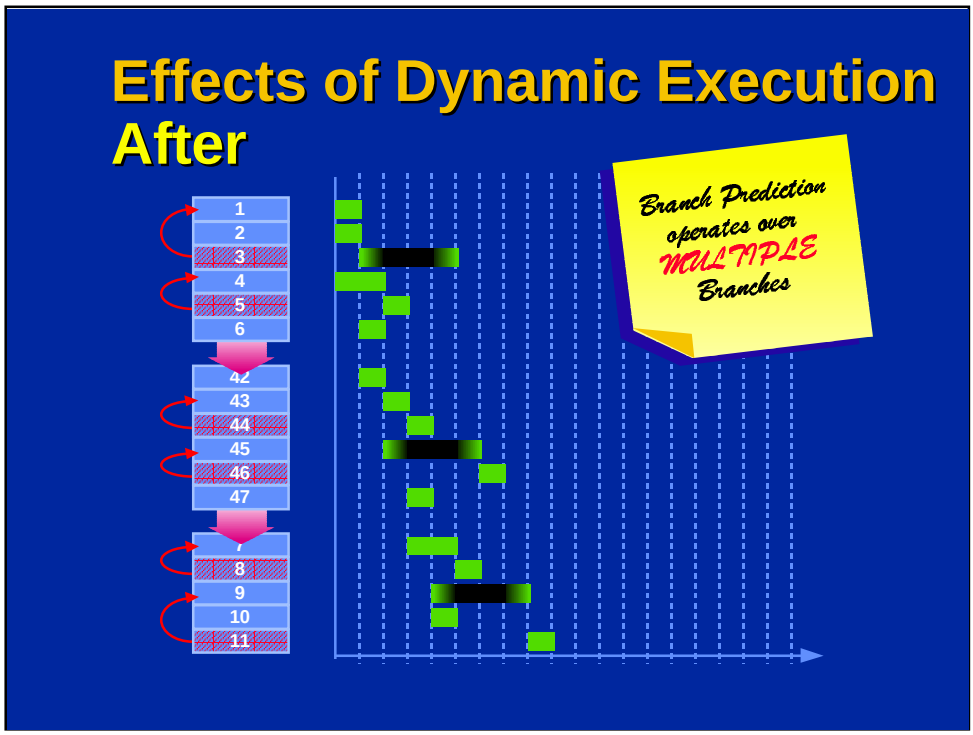
-Speculative Execution

Rather than waiting the P6 speculatively executes instructions after the one that's stalled. That is, rather than doing nothing, the P6 goes and starts some other work since we'll need it anyway. The P6 ensures that data dependancies are observed (i.e. cannot arbitrarily execute any instructions), ensuring that it is doing the right work.

The P6 must also be able to easily unravel this speculative work done in case an interrupt, mis-predicted branch, page-fault or debug trap occurs.



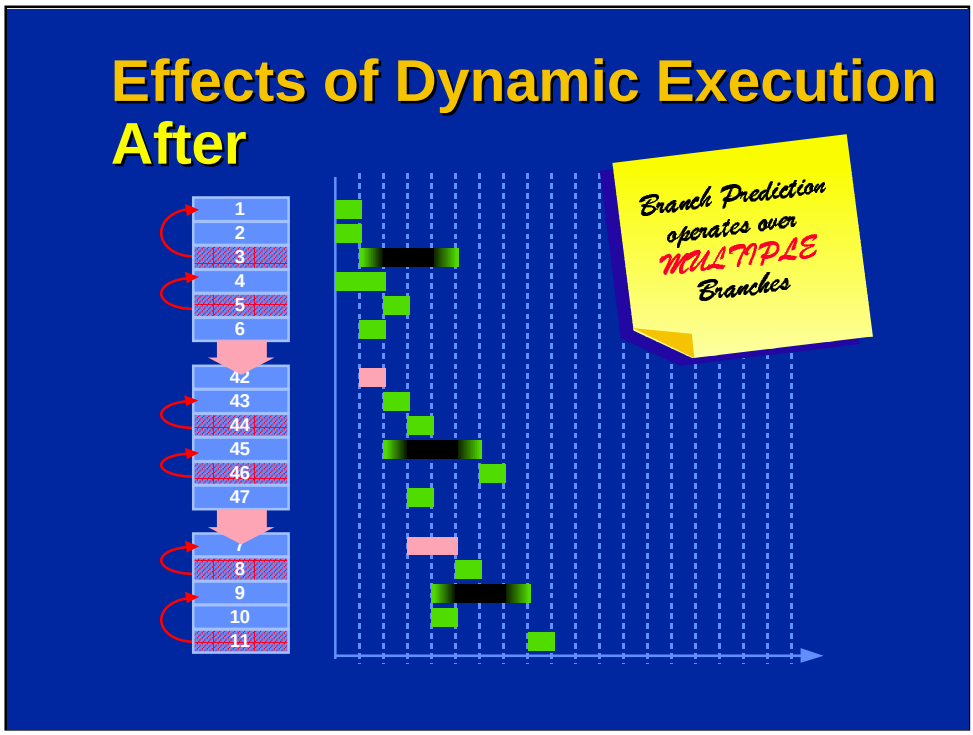
The red boxes (green on the previous slide) indicate those that are speculatively executed.



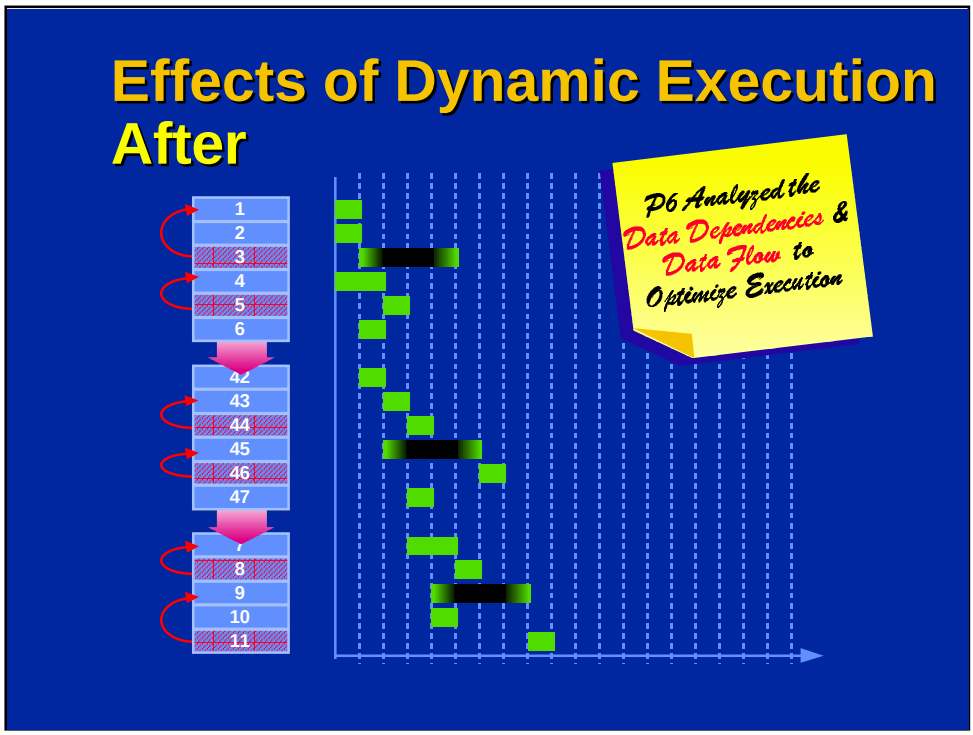
The “front end” of the P6 must aggressively get work (i.e. a large array of instructions) for the core to choose from.

On average, there is a branch every 5 to 7 instructions, so the P6 must be able to accurately preduct multiple branches.

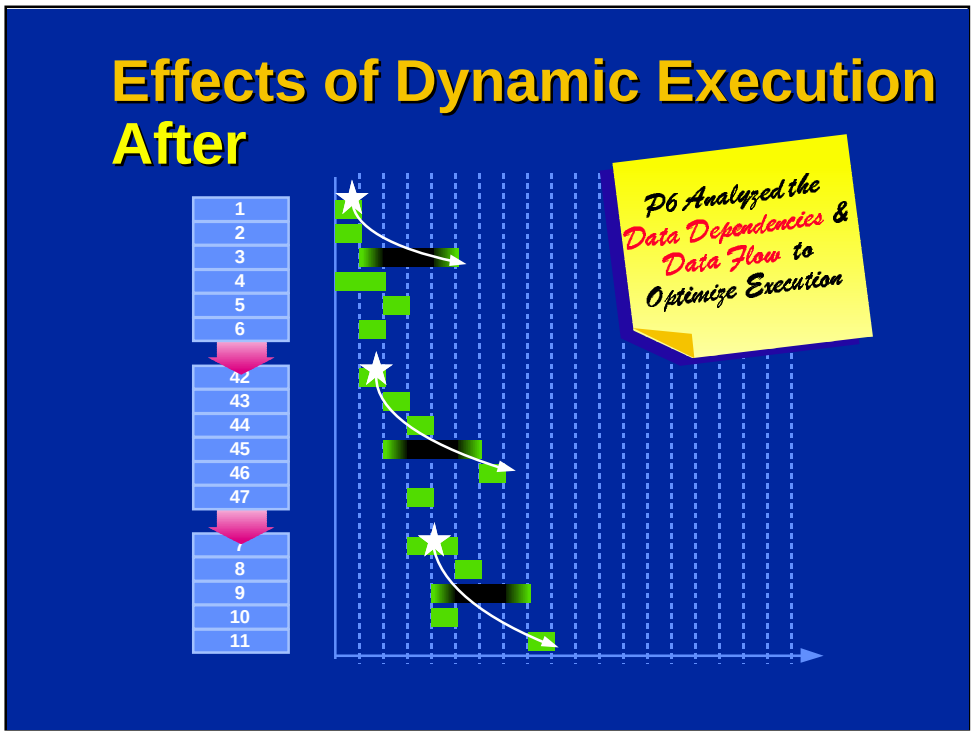
The P6 contains a large Branch Target Buffer which uses both a static algorithm and a history based algorithm to predict branches correctly over 90% of the time.



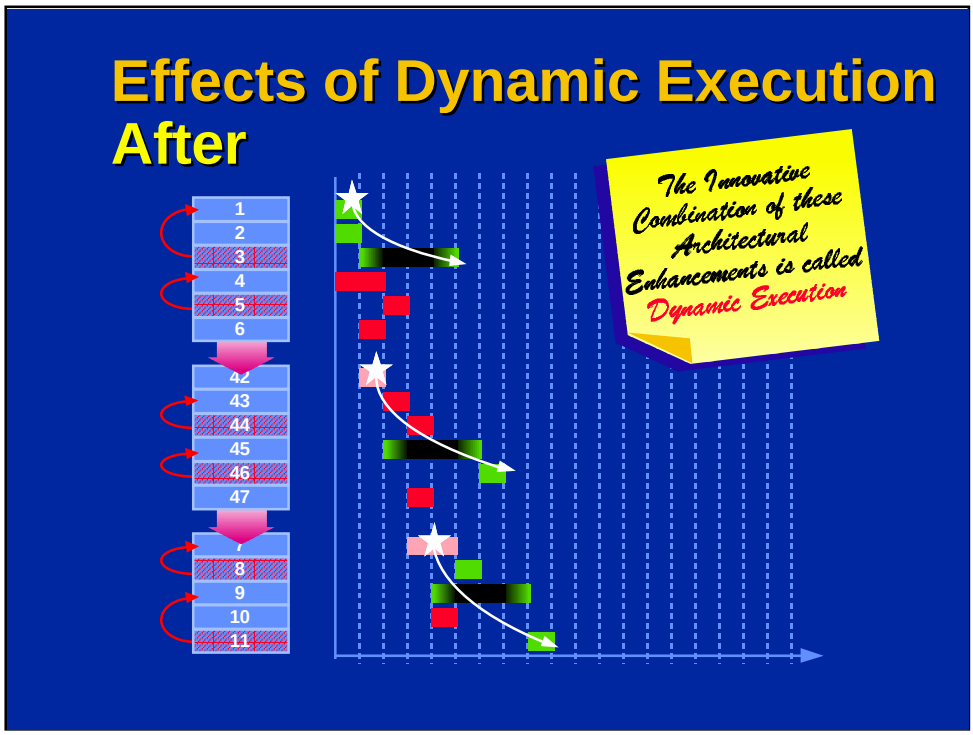
The pink arrows and instructions above highlight the P6's branch prediction over multiple branches.



Again, the small red arrows indicate the P6's analysis of data dependencies and data flow to determine the most efficient order in which to execute instructions.



The benefit of this is that the P6 can make forward progress at multiple points within a program flow (compared with the Pentium® processor that only has a single point of forward progress).



In sum, Dynamic Execution is the unique combination of:

Improved Branch Prediction	(get lots of work to do)
Speculative Execution	(ability to execute in <u>any</u> order)
and	
Data Flow Analysis	(choose the best order)