1. Pipelines: Pentium-Prozessor

a) (1) Doppelt so schnelle Integer-Ausführung, fünf mal so schnelle Floating-Point-Ausführung;
(2) Befehlsdekodierung in einem Takt (statt zwei beim i486);
(3) superskalar mit zwei parallelen Pipelines;
(4) Verbesserte Branch-Execution bei „taken branches“ dank „branch target buffer“ (BTB)
(5) separate Code- und Daten-Caches (beim i486: ein gemeinsamer Cache)
... (nur nach drei Verbesserungen gefragt)

b) „Logic in D1 [erste Decode-Phase] ensures that the source and destination registers of the instruction issued to the V pipe differ from the destination register of the instruction issued to the U pipe. This arrangement eliminates read-after-write (RAW) and write-after-write (WAW) dependencies. Write-after-read (WAR) dependencies need not be checked because redds occur in an earlier stage of the pipelines than writes. “ [1]

c) Pentium-Integer-Pipeline: PF, D1, D2, E, WB (Prefetch, Decode1, Decode2, Execute, Write result): D1 macht aus einer Instruktion eines oder mehrere Kontrollwörter. D2 dekodiert ein Kontrollwort.

Pentium-Floating-Point-Pipeline: PF, D1, D2, E, X1, X2, WF, ER (Prefetch; Decode1: Decode2; E = Operand Fetch: Zugriff auf Daten-Cache und FP-Register, um Operanden bereit zu stellen; X1+X2: Ausführen, FW = Write Float: Rechnung beenden und Ergebnis in FP-Register schreiben; ER = Error Reporting)

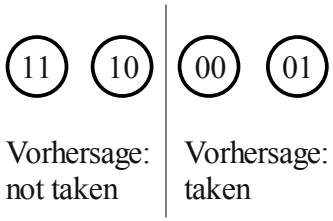
d) „The reason is that the superscalar design and branch prediction demand more bandwidth than a unified cache similar to that of the i486 CPU can provide. (1) Efficient branch prediction requires that the destination of a branch be accessed simultaneously with data references of previous instructions executing in the pipeline. (2) The parallel execution of data memory references requires simultaneous accesses for loads and stores. (3) In the context of the overall Pentium microprocessor design, handling self-modifying code for separate code and data caches is only marginally more complex than for a unified cache. “ [1]

[1] D. Alpert, D. Avnon: Architecture of the Pentium Microprocessor , IEEE Micro, Juni 1993 (11-21)

2. Superskalarität: Prädiktoren

a) z. B. YesNo = 11111100000011111000000.... Die Historie merkt sich nur die letzten vier Sprungentscheidungen und hat damit keine Chance zu erkennen, ob bereits 4-, 5- oder 6-mal gesprungen wurde.

b)		nnn	nnt	ntt	ttn	tnn
T	NNN	00 :)	00	00	00	00
T	NNT	**01**	00 :)	00	00	00
N	NTT	01	**01**	00 :(	00	00
N	TTN	01	01	**10**	00 :(	00
T	TNN	01	01	10	**10**	00 :)
T	NNT	01	01 :)	10	10	**01**
N	NTT	01	01	10 :)	10	01
N	TTN	01	01	**11**	10 :)	01
T	TNN	01	01	11	**11**	01 :)
T	NNT	01	01 :)	11	11	01
N	NTT	01	01	11 :)	11	01
N	TTN	01	01	11	11 :)	01



c) Nein – aus demselben Grund, aus dem die vierstufige Historie nicht für 111111000000... ausreicht: Die einstufige Historie kann nicht unterscheiden, ob zuletzt 1-mal oder 2-mal gesprungen wurde. Da hilft auch das zweite Prädiktor-Bit nicht.