Rechnerarchitektur 1

Inhaltsübersicht

Rechnerarchitektur 1

Literatur

Hennessy, John L.; Patterson, David A. (Standardwerk)

Computer architecture : A quantitative approach

2nd ed 1995. hardback £25.00

Morgan Kaufmann

1-55860-329-8

Giloi, Wolfgang K. (empfehlenswert, Übersicht)

Rechnerarchitektur

2.vollst. überarb. Aufl.1993 DM58.00

Springer-Verlag

3-540-56355-5

Hwang, Kai (advanced, viel parallele Arch.)

Advanced computer architecture : Parallelism, scalability and programmability

1993. 672 pp. paperback £24.99

McGraw-Hill ISE

0-07-113342-9

Patterson, David A.; Hennessy, John L. (leichter Einstieg)

Computer organization and design : The hardware/software interface

2nd ed 1994. hardback £60.50

Morgan Kaufmann

1-55860-281-X

Stone, Harold S. (optional, Speicherarchitekturen)

High performance computer architecture

3rd ed 1990. 736 pp. hardback £39.50

Addison-Wesley USA

0-201-52688-3

Tanenbaum, Andrew S. (optional, Standardwerk für Betriebssysteme))

Modern operating systems

1992. 1055 pp. paperback £26.95

Prentice Hall US

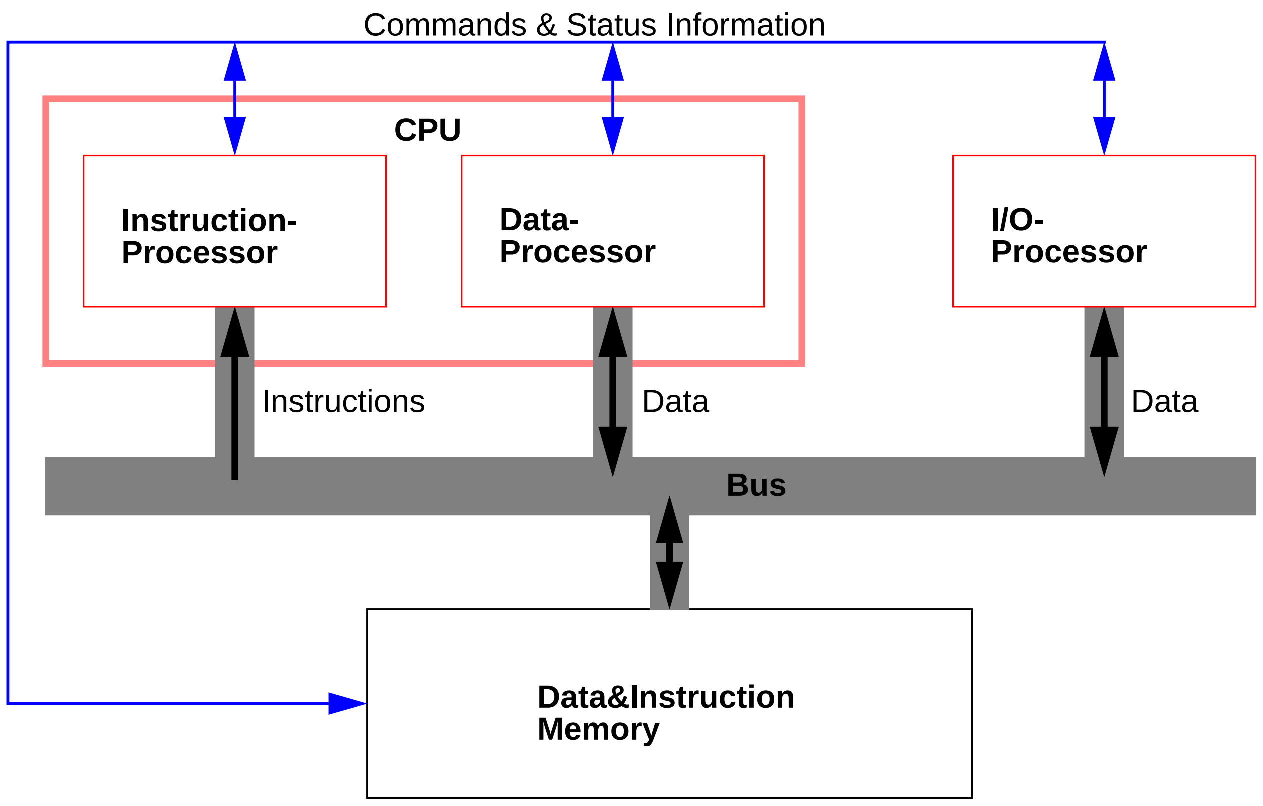
0-13-595752-4

Einleitung

von Neumann Architektur:

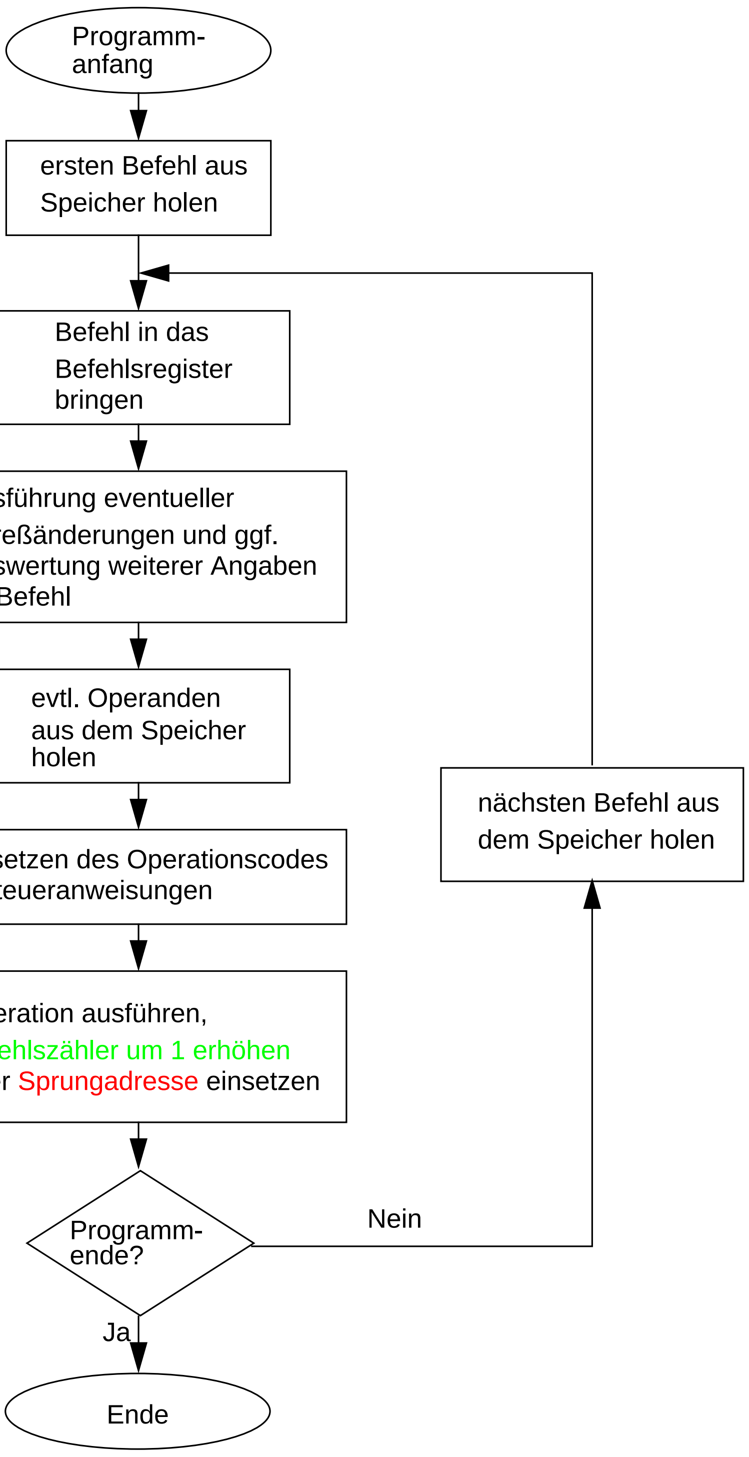
Beschreibt eine abstrakte Maschine des minimalen Hardwareaufwandes, bestehend aus:

- *einer zentralen Recheneinheit (CPU), die sich in Daten- und Befehlsprozessor aufteilt*
- *einem Speicher für Daten und Befehle*
- *einem Ein/Ausgabe-Prozessor zur Anbindung periphärer Geräte*
- *einer Verbindungseinrichtung (Bus) zwischen diesen Komponenten*



von Neumann Architektur

Verarbeitung von Instruktionen:


Programm-
anfang
ersten Befehl aus Speicher holen
Befehl in das Befehlsregister bringen
Ausführung eventueller Adreßänderungen und ggf. Auswertung weiterer Angaben im Befehl
evtl. Operanden aus dem Speicher holen
Umsetzen des Operationscodes in Steueranweisungen
Operation ausführen, Befehlszähler um 1 erhöhen oder Sprungadresse einsetzen
Programm-
ende?
Nein
Ja
Ende
nächsten Befehl aus dem Speicher holen

History of Computer Generations

History of computer generations

Generation	Technology & Architecture	Software & Applikations	Systems
First (1945-54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed point arithmetic	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU	ENIAC, Princeton IAS, IBM 701
Second (1955-64)	Discrete transistors and core memories, floating point arithmetic, I/O processors, multiplexed memory access	HLL used with compilers, subroutine libraries, batch processing monitor	IBM 7090, CDC 1604, Univac LARC
Third (1965-74)	Integrated ciruits (S/MSI), microprogramming, pipelining, cache, lookahead processors	Multiprogramming and timesharing OS, multiuser applications	IBM 360/70, CDC 6600, TI-ASC, PDP-8
Fourth (1975-90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers	Multiprocessor OS, languages, compilers and environment for parallel processing	VAX 9000, Cray X-MP, IBM 3090
Fifth (1991-96)	ULSI/VHSIC processors, memory and switches, high density packaging	Massively parallel processing, grand challenge applications	Cray MPP, CM-5, Intel Paragon
Sixth (present)	scalable off-the-shelf architectures, workstation clusters, high speed interconnection networks	heterogeneous processing, fine grain message transfer	Gigabit Ethernet, Myrinet

[Hwang, Advanced Computer Architecture]

Was ist Rechnerarchitektur ?

Rechnerarchitektur <- ’ computer architecture ’

The types of architecture are established not by architects but by society, according to the needs of the different institutions. Society sets the goals and assigns to the architect the job of finding the means of achieving them.

Entsprechend ihrem Zweck unterscheidet man in der Baukunst Typen von Architekturen .....

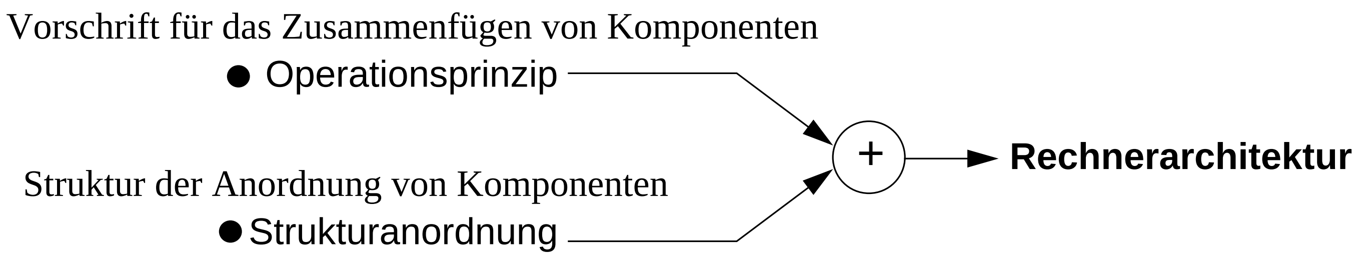
Arbeit des Computer-Architekten:

- finden eines Types von Architektur, die den vorgegebenen Zweck erfüllt.
- Architekt muß gewisse Forderungen erfüllen.
  - Leistung
  - Kosten
  - Ausfalltoleranz
  - Erweiterbarkeit

Materialien des Gebäudearchitekten : Holz, Stein, Beton, Glas ...

Materialien des Rechnerarchitekten : integrierte Halbleiterbausteine ...

Folgende Komponenten stellen die wesentlichen HW-Betriebsmittel des Computers dar: Prozessoren, Speicher, Verbindungseinrichtungen


Vorschrift für das Zusammenfügen von Komponenten
● Operationsprinzip
Struktur der Anordnung von Komponenten
● Strukturanordnung
+
Rechnerarchitektur

Bestandteile der Rechnerarchitektur

Eine Rechnerarchitektur ist bestimmt durch ein Operationsprinzip für die Hardware und die Struktur ihres Aufbaus aus den einzelnen Hardware-Betriebsmitteln. [Giloi 93]

Das **Operationsprinzip**

Das Operationsprinzip definiert das funktionelle Verhalten der Architektur durch Festlegung einer Informationsstruktur und einer Kontrollstruktur.

Die **(Hardware) - Struktur**

Die Struktur einer Rechnerarchitektur ist gegeben durch Art und Anzahl der Hardware-Betriebsmittel sowie die sie verbindenden Kommunikationseinrichtungen.

- Kontrollstruktur : Die Kontrollstruktur einer Rechnerarchitektur wird durch Spezifikation der Algorithmen für die Interpretation und Transformation der Informationskomponenten der Maschine bestimmt.
- Informationsstruktur : Die Informationsstruktur einer Rechnerarchitektur wird durch die Typen der Informationskomponenten in der Maschine bestimmt, der Repräsentation dieser Komponenten und der auf sie anwendbaren Operationen. Die Informationsstruktur läßt sich als Menge von 'abstrakten' Datentypen spezifizieren.
- Hardware-Betriebsmittel : Hardware-Betriebsmittel einer Rechnerarchitektur sind Prozessoren, Speicher, Verbindungseinrichtungen und Peripherie.
- Kommunikationseinrichtungen : Kommunikationseinrichtungen sind die Verbindungseinrichtungen, wie z.B.: Busse, Kanäle, VNs ; und die Protokolle, die die Kommunikationsregeln zwischen den Hardware-Betriebsmittel festlegen.

Unterscheidung der Parallelrechner nach dem Operationsprinzip

Unter dem Operationsprinzip versteht man das funktionelle Verhalten der Architektur, welches auf der zugrunde liegenden Informations- und Kontrollstruktur basiert.

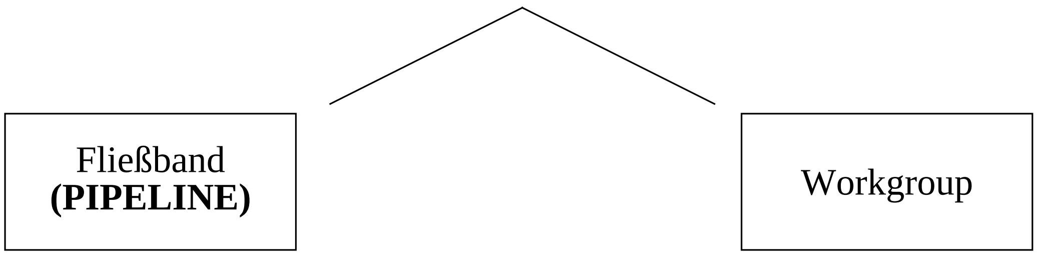
Klassifikation nach Flynn (recht grob)

SISD	single instruction - single data stream
SIMD	single instruction - multiple data stream
MISD	multiple instruction - single data stream
MIMD	multiple instruction - multiple data stream

Unterscheidung der Parallelrechner nach dem Operationsprinzip

Beispiel : Automobilfertigung

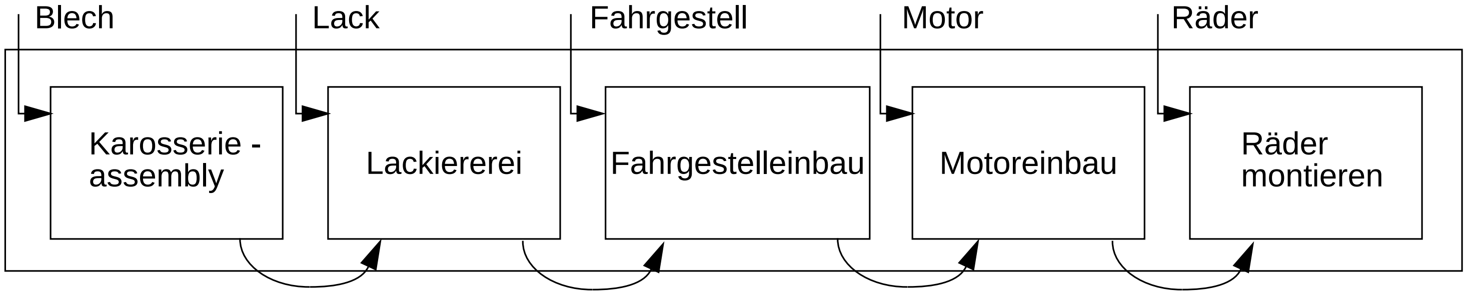
Welche Möglichkeiten des Zusammenfügens (Assembly) gibt es ?



Worauf muß man achten ?

Abhängigkeiten !

Fließband



Produktion von verschiedenen Modellen : 3 Farben R(ot), G(rün) B(lau)

2 Motoren N(ormal), I(njection)

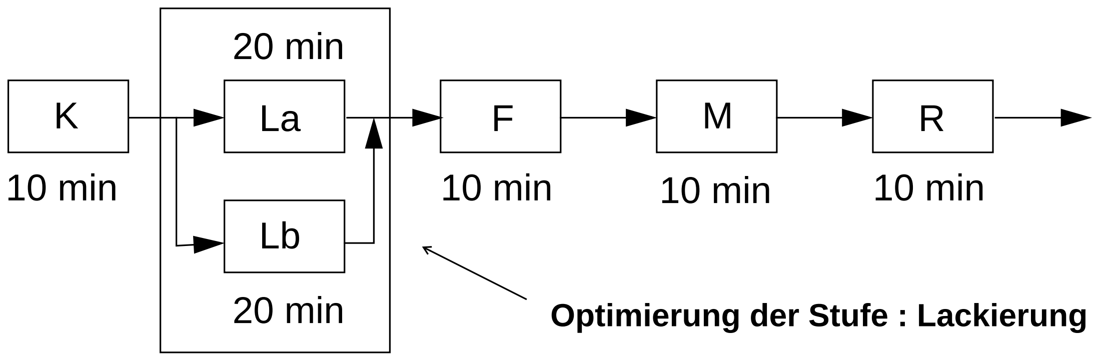
2 Karosserien L(imousine), K(ombi)

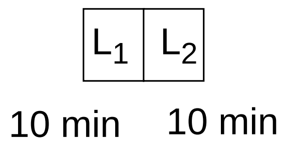
41

Auftrag	
F	R
M	N
K	K

Unterscheidung der Parallelrechner nach dem Operationsprinzip

Pipeline des Fertigungsvorgangs





Stufen - Zeit - Diagramm der Pipeline

stage ⟶

Auftrag 41

K	L_1	L_2	F	M	R	
41						1 : 5
42	41					2 : 4
43	42	41				3 : 3
	43	42	41			3 : 3
		43	42	41		3 : 3
			43	42	41	3 : 3
				43	42	2 : 4
					43	1 : 5

time ↓

’ von Neumann ’ - Rechner (Burks, Goldstine, von Neumann)

Architektur des minimalen Hardwareaufwands !

Die hohen Kosten der Rechnerhardware in der Anfangszeit der Rechnerentwicklung erzwang den möglichst kleinen Hardwareaufwand.

- Befehle holen	
- Operand holen (2x)	Indexoperation, Adressrechnung
- Befehle interpretieren	
- Befehl ausführen	arithmetische Op. (gr. Aufwand an Zeit + HW)

Ist die arithmetische Operation der wesentliche Aufwand, so können die anderen Teilschritte sequentiell ausgeführt werden, ohne die Verarbeitungszeit stark zu verlängern.

Zeitsequentielle Ausführung der Befehle auf minimalen Hardware-Resourcen

Veränderung der Randbedingungen :

1. Reduzierung der Hardwarekosten
   Hochintegration, Massenproduktion
2. Aufwand in den einzelnen Op. verschob sich.

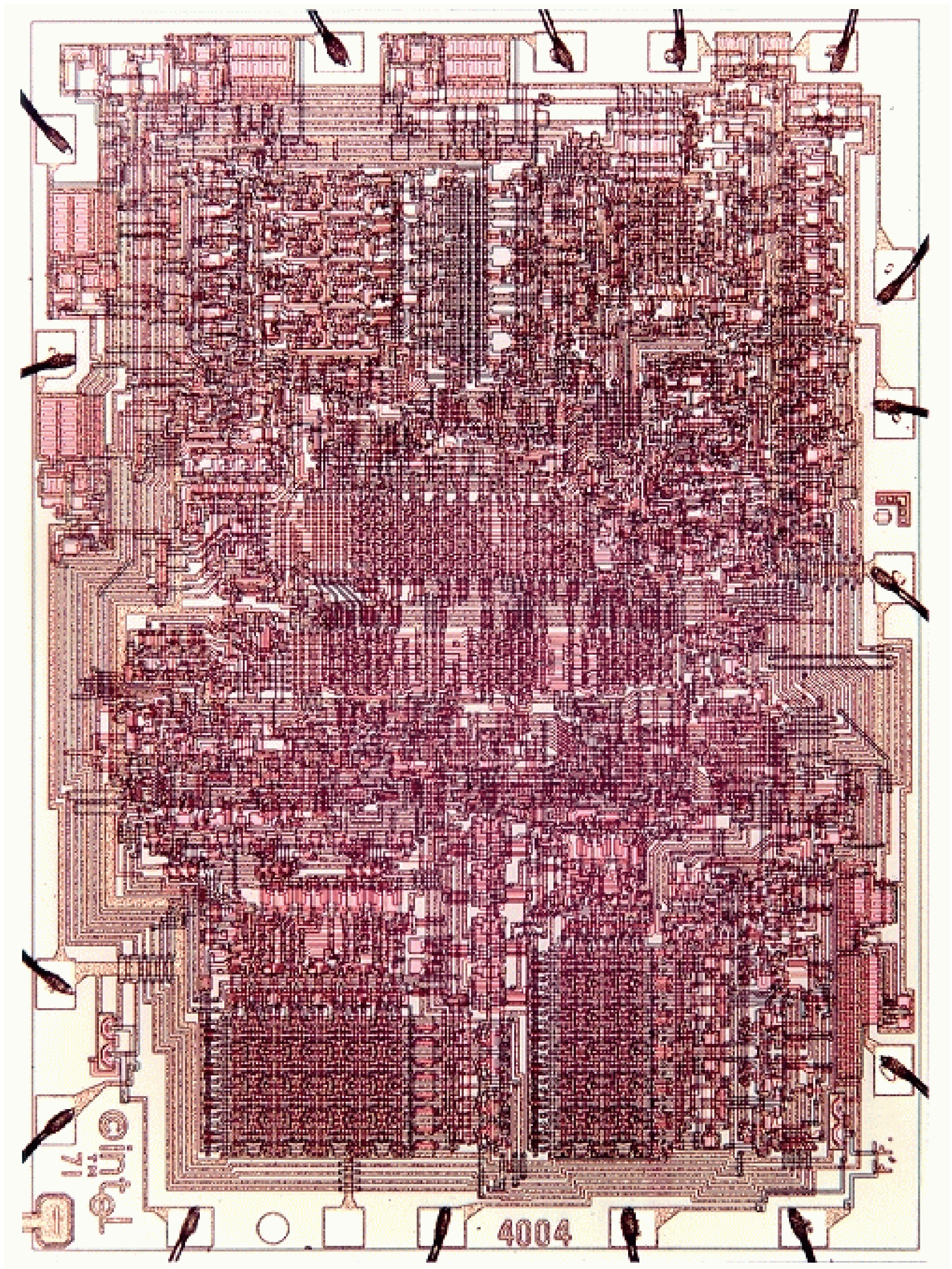
Gesucht : Ideen, mit zusätzlicher Hardware bisher nicht mögliche Leistungssteigerungen zu erreichen.

Ideen = Operationsprinzipien

Technology

Introduction

First integrated microprocessor Intel 4004 1971 2700 transistors

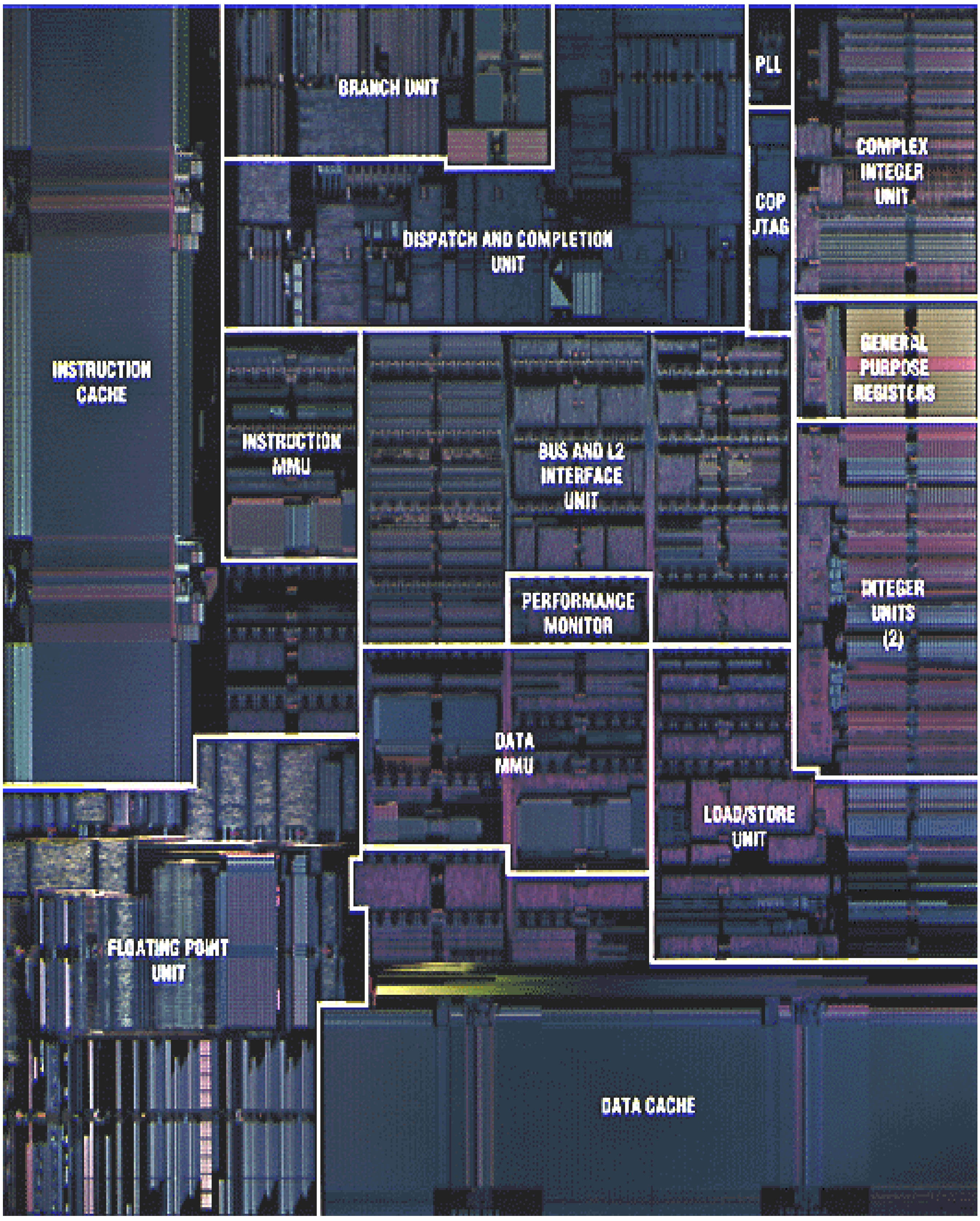


Technology

Introduction

year 1998 microprocessor PowerPC 620 7,5 Mio. transistors

Motorola's PowerPC™ 620 32/64-Bit RISC Microprocessor



Technology

Technology forecast

The 19th April 1965 issue of Electronics magazine contained an articel with the title "Cramming more components onto integrated circuits" Its author, Gorden E. Moore, director, Research and Development, Fairchild Semiconductor, had been asked to predict what would happen over the next ten years in the semiconductor component industry. His article speculated that by 1975 it would be possible to cram as many as 65 000 components onto a single silicon chip about 6 mm^2. [Spectrum, June ’97]

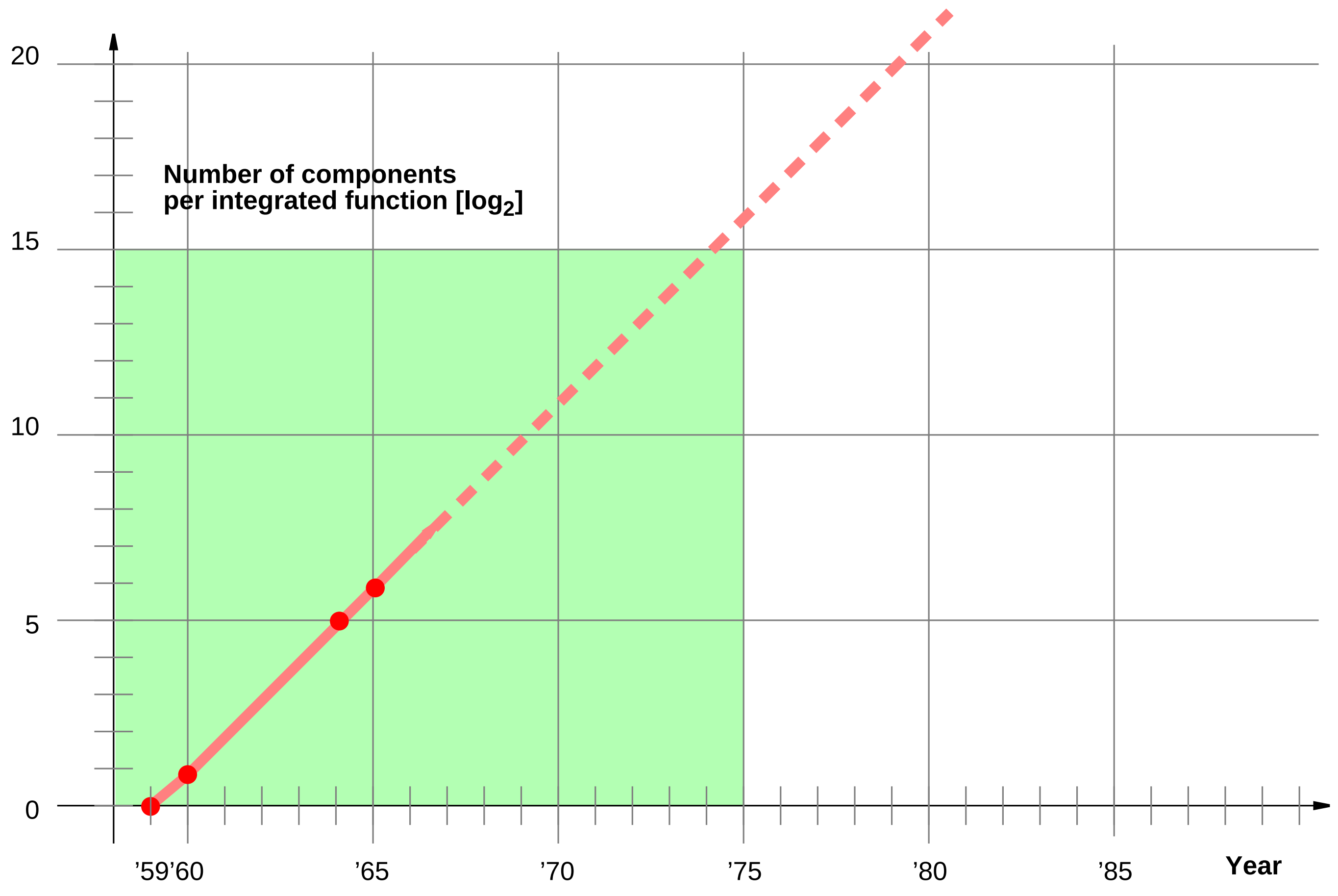
Moore based his forecast on a log-linear plot of device complexity over time.

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain constant for at least 10 years.

Moore’s astonishing prediction was based on empirical data from just three Fairchild data points!

- first planar transistor in 1959
- few ICs in 1960 with ICs in production with 32 components in 1964
- IC with 64 components in 1965

In 1975 he revisted the slope of his function to doubling transistor count every 18 month and this stayed true until today.



Technology

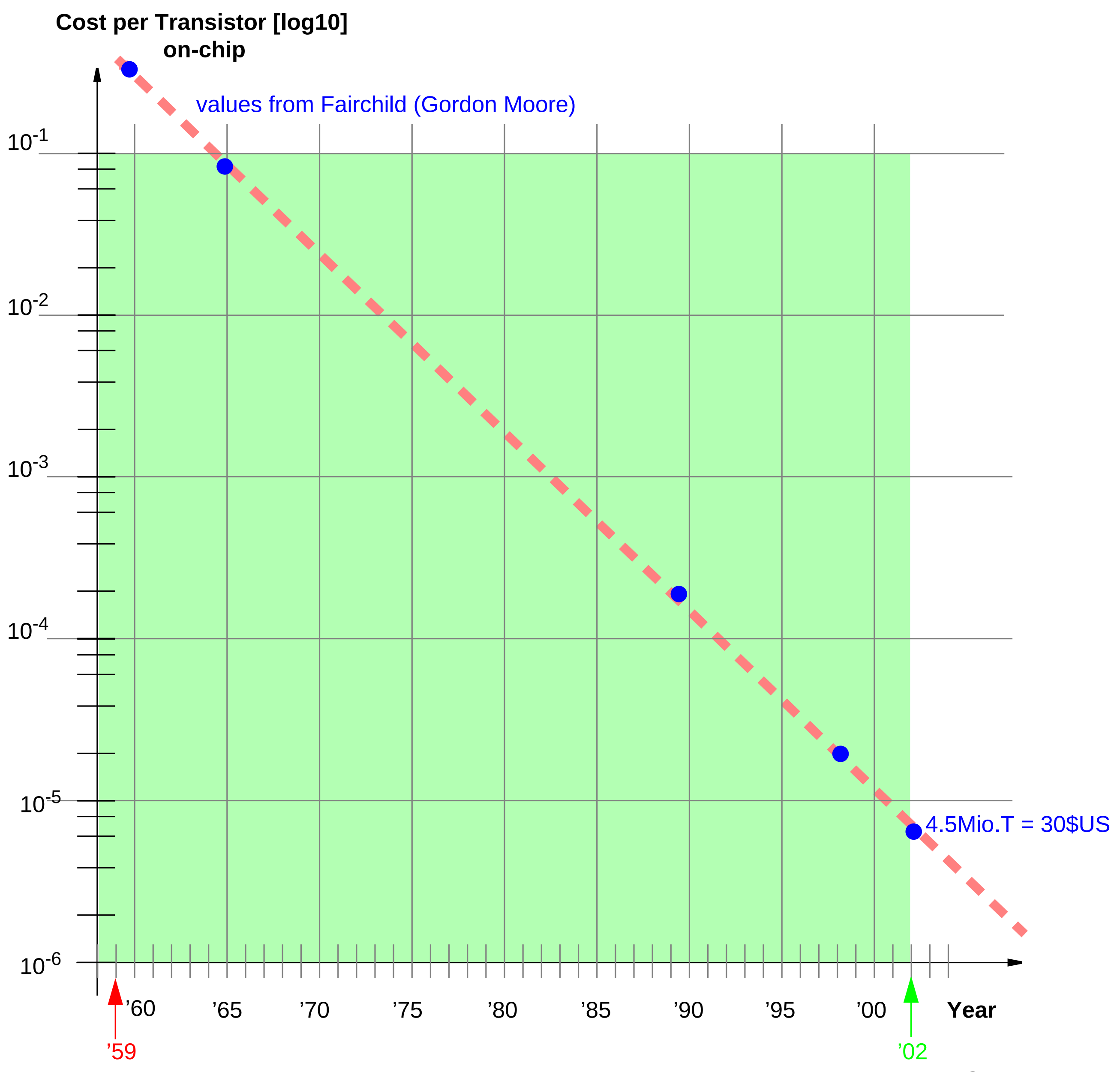
Andrew (Andy) Grove **Robert Noyce** **Gordon Moore**

[2] Gordon Moore [right] relaxes with fellow pioneers of the electronic age: Robert Noyce [center] and Andrew Grove [left]. Moore and Noyce contributed to the development of the planar IC. Grove is now president and chief executive officer of Intel Corp.

Technology

Cost reduction due to mass fabrication

red curve for logic chips
(no memory chips)



~ 6.6 * 10^{-6} $US

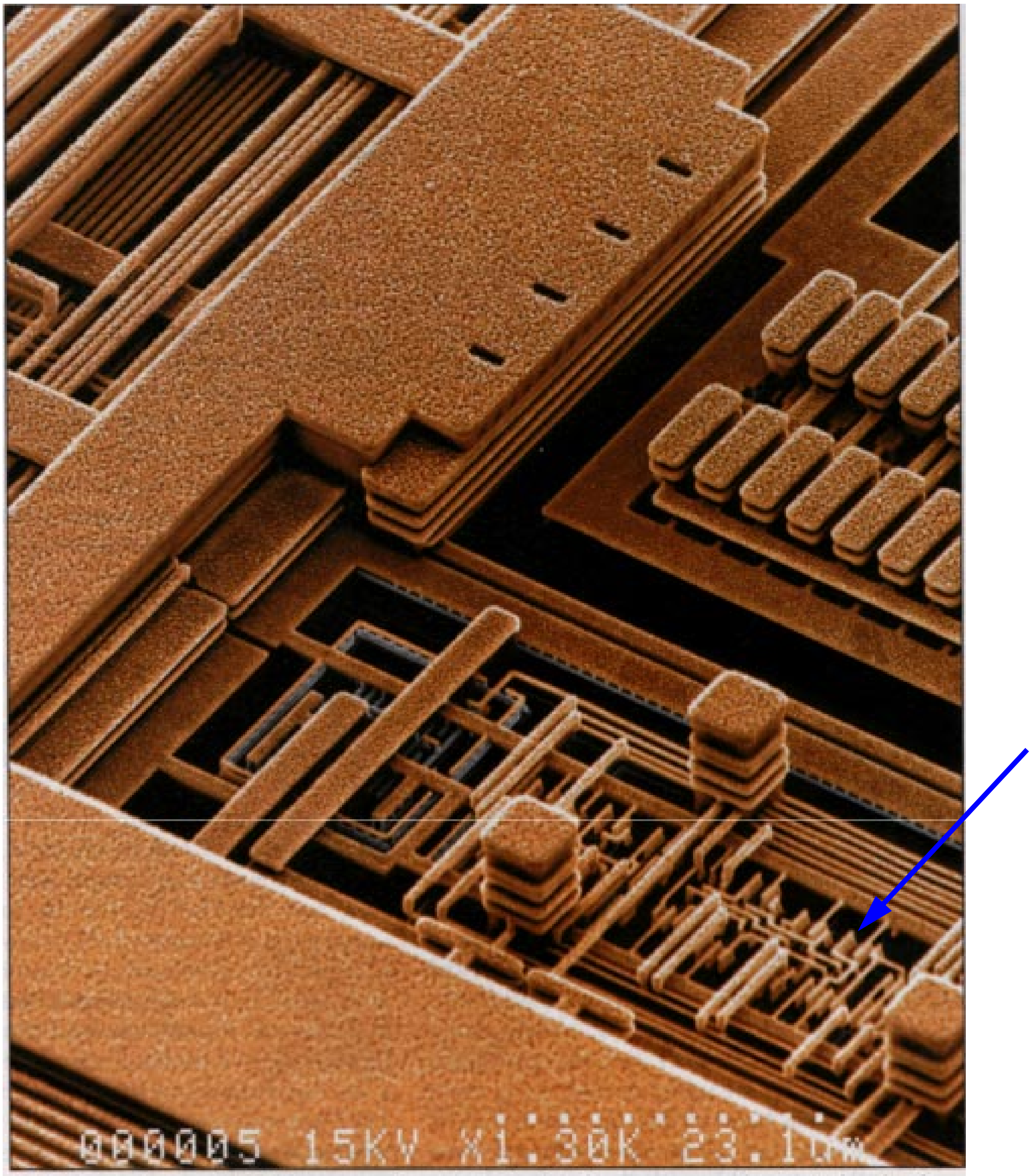
year 2002: 4.5Mio. transistors in 0.18µm CMOS technology on a 5x5mm die with BGA package ~ 30 $US = 6.6 x 10^{-6} $US per transistor; standard cell design

Technology

Modern Chip Technology

Using copper for the 6 metal interconnect structure of a CMOS chip delivers lower ressitance of the wires and thus increases the performance.

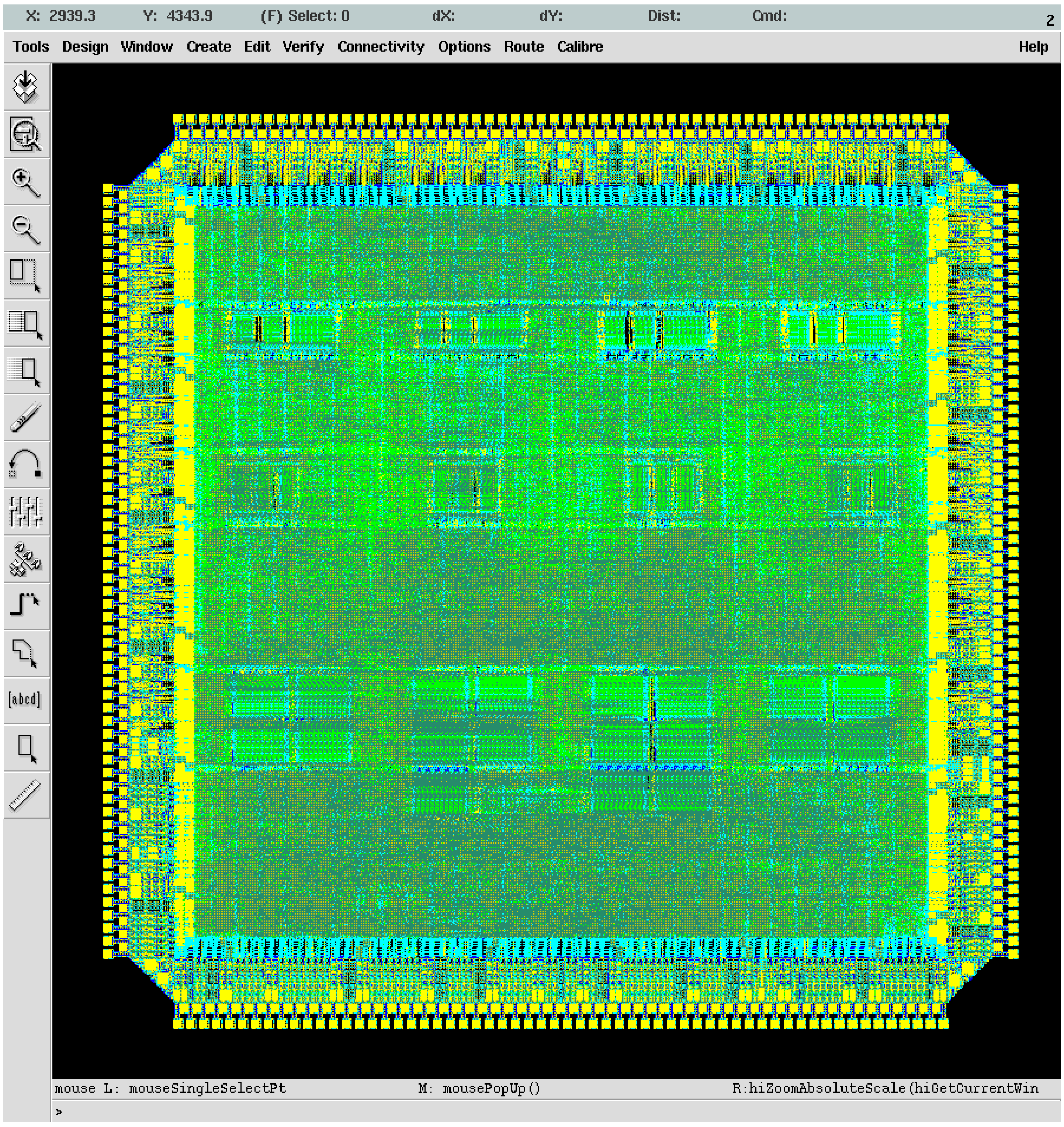
Gate structures can be found in the lower right part of the picture (arrow)



[1] IBM Corp.'s new CMOS 7S process for manufacturing ICs uses copper for its six levels of interconnections, and has effective transistor channel-lengths of only 0.12 µm. It is the first commercial fabrication process to use copper wires [see "The Damascus connection," p. 25].

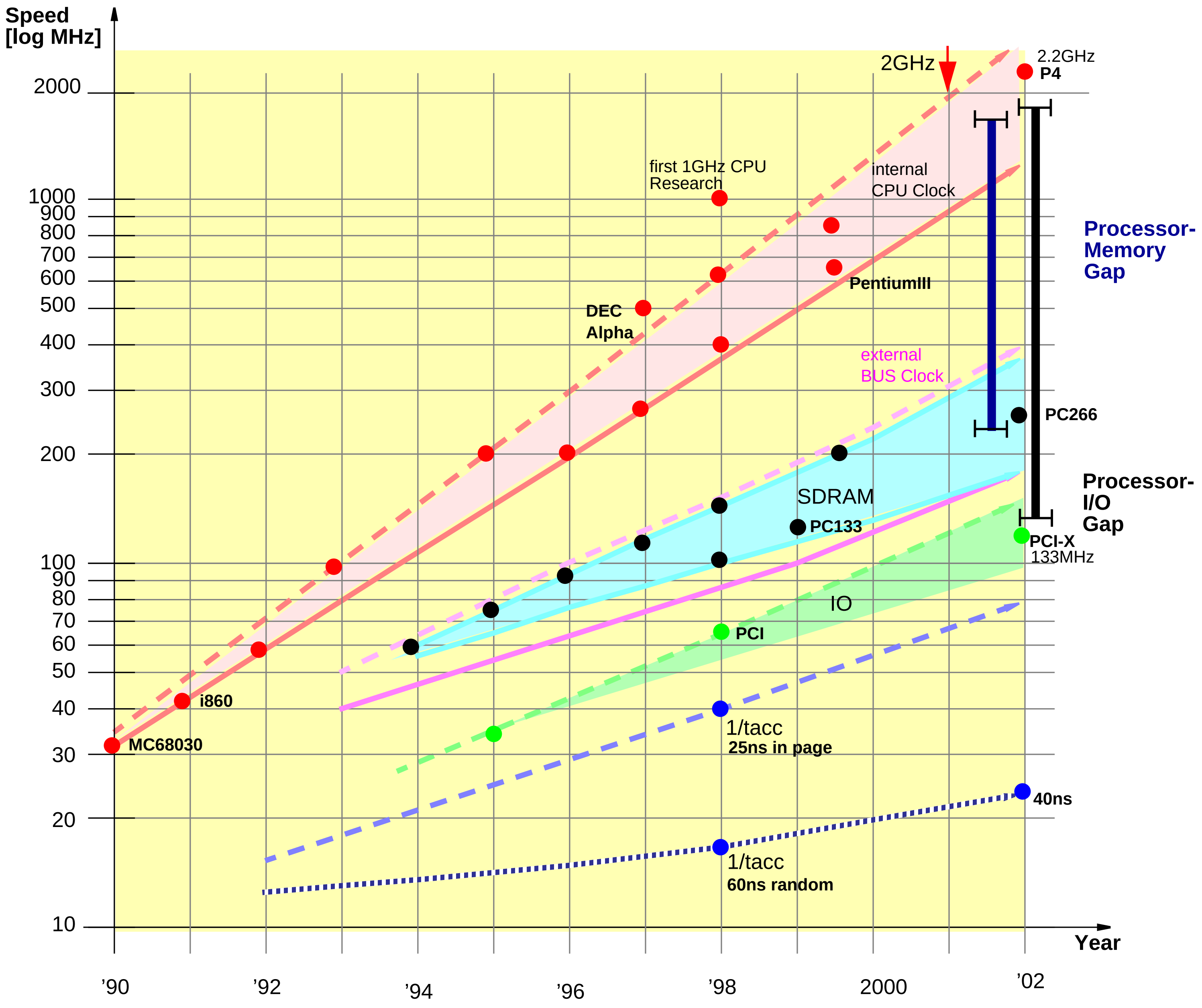
ATOLL - ATOmic Low Latency

- approx. 5 Mio transistors (1.4 Mio logic gates)
- Die area: 5.78mm x 5.78mm
- 0.18um CMOS (UMC, Taiwan)
- 385 staggered I/O pads
- 3 years of development



Speed Trends

Processor Memory I-O -Speed Trends



Die ständige Steigerung der Rechenleistung moderner Prozessoren führt zu einer immer größer werdenden Lücke zwischen der Verarbeitungsgeschwindigkeit des Prozessors und der Zugriffsgeschwindigkeit des Hauptspeichers. Die von Generation zu Generation der DRAMs steigende Speicherkapazität (x 4) führt auf Grund der 4-fachen Anzahl an Speicherzellen trotz der Verkleinerung der VLSI-Strukturen zu nur geringen Geschwindigkeitssteigerungen.

(not included: new DRAM technology RAMBUS)

Speed Trends

Processor Memory I-O -Speed Trends

Results

- a high Processor-Memory Performance Gap (increasing)
- a very high Processor-I/O Performance Gap (increasing)

Single Processor Performance will increase

Number of processors in SMPs will increase moderately

STOP Bus Systems will become the main bottleneck

Solution

- Transition to “switched” Systems

  on all system levels

  - Memory
  - I/O
  - Network

Switched System Interconnect

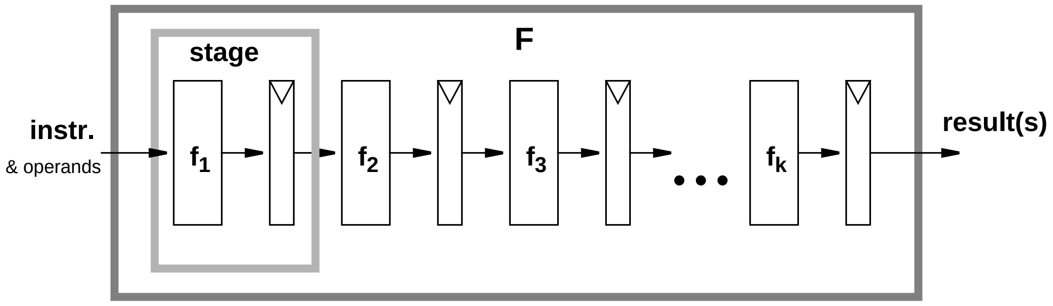
Pipelining

The performance gain achieved by pipelining is accomplished by partitioning an operation F into multiple suboperations f_1 to f_k and overlapping the execution of the suboperations of multiple operations F_1 to F_n in successive stages of the pipeline [Ram77]

Assumptions for Pipelining

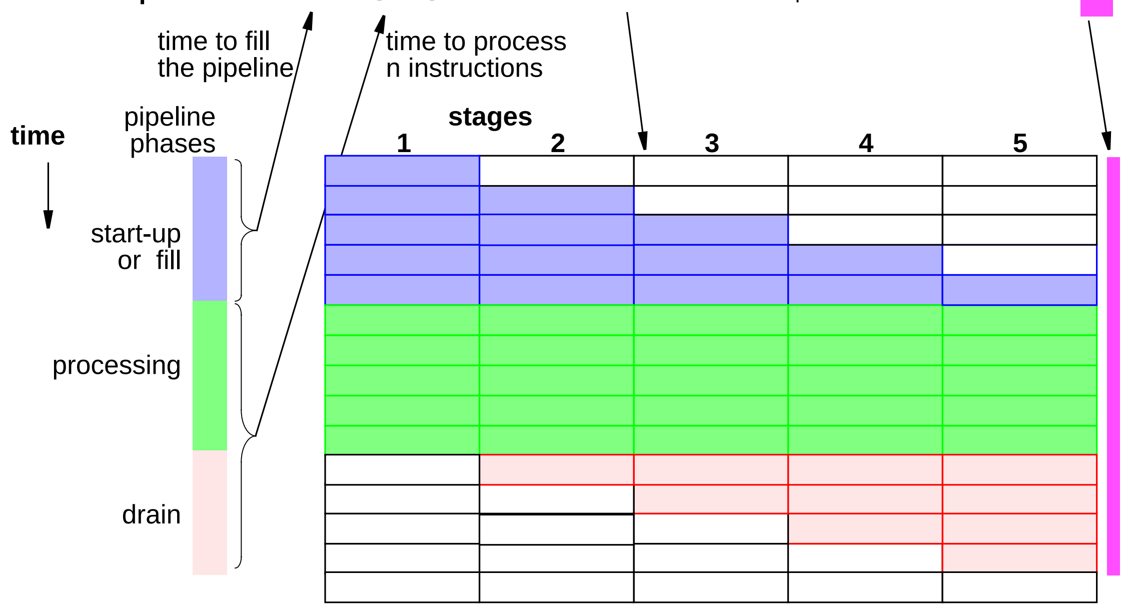
1. the operation F can be partitioned
2. all suboperations f_i require approximately the same amount of time
3. there are several operations F the execution of which can be overlapped
4. the execution time of the suboperations is long compared with the register delay time

Linear Pipeline with k Stages



Pipelined Operation Time

$t_p\,(n, k) = k + (n-1)$ for this example: $t_p\,(10,5) = 5 + (10 - 1) = 14$



Durchsatz Throughput

$$TP\,(n, k) = \frac{\text{number of operations}}{t_p\,(n,k)} = \frac{n}{k + (n-1)} \quad \left[\frac{\text{operations}}{\text{time unit}}\right]$$

Gewinn Gain

$$S\,(n, k) = \frac{\text{scalar execution time}}{\text{pipelined execution time}} = \frac{n * k}{k + (n-1)} \qquad \lim_{n\to\infty} S \to k$$

Effizienz Efficiency

initiation rate, latency

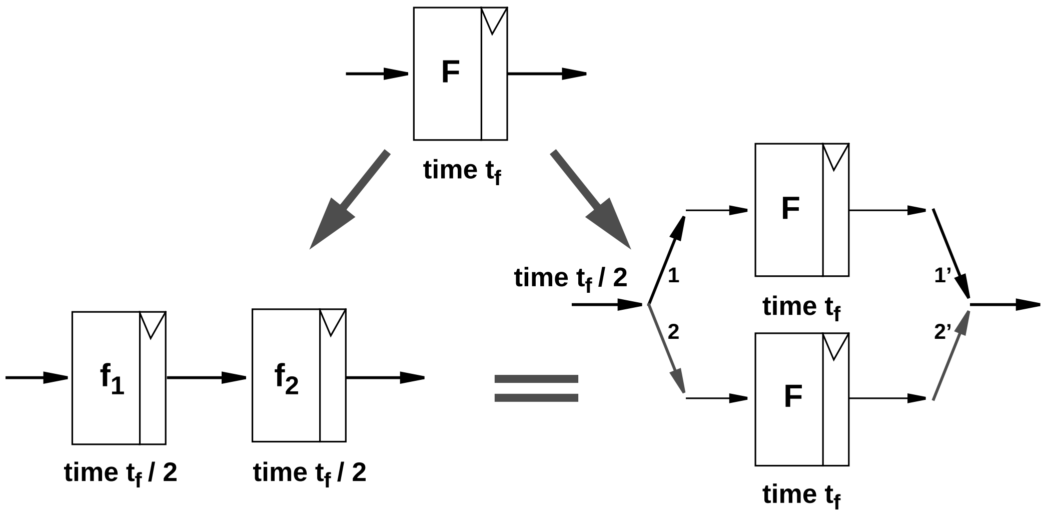
$$E\,(n, k) = \frac{1}{k} * S\,(n, k) = \frac{n * k}{k * (k + (n-1))} = \frac{n}{k + (n-1)}$$

Pipeline Interrupts

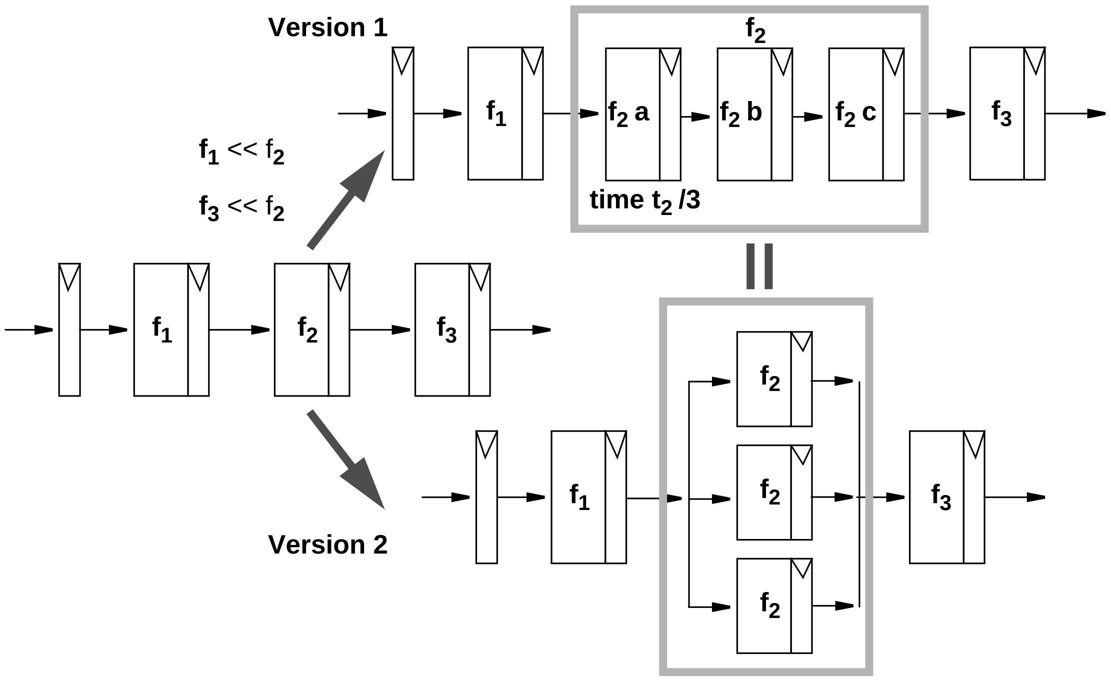
- data dependencies
- control-flow dependencies
- resource dependencies

Assumptions for Pipelining

1 the operation F can be partitioned



2 all suboperations f_i require approximately the same amount of time



Assumptions for Pipelining

3 there are several operations F the execution of which can be overlapped

If there is a discontinuous stream of operations F owing to a conflict, bubbles are inserted into the pipeline. This reduces the performance gain significantly.

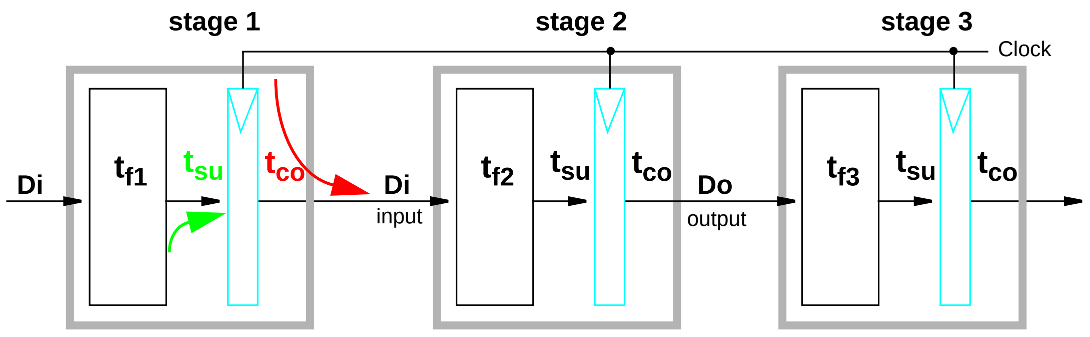
A typical example of this is the control dependency of the instruction pipeline of a processor. Here, each conditional branch instruction may disrupt the instruction stream and cause (k-1) bubbles (no-operations) in the pipeline, if the control flow is directed to the nonpredicted path.

4 the execution time of the suboperations is long compared with the register delay time

Assuming a division of the operation F into three suboperations f_1, f_2, f_3, and also no pipelining, the operation F can be executed in the time:

$$t(F) = t_{f1} + t_{f2} + t_{f3}$$

Introduction of registers



$$t(F) = \underbrace{(\max(t_{fi}) + t_{co} + t_{su})}_{t_{cyc}} * \underset{\text{k stages}}{3} = 3 * \max(t_{fi}) + 3 * \underbrace{(t_{co} + t_{su})}_{\text{register delay time}}$$

$$t_{cyc} = \max(t_{fi}) + t_{co} + t_{su} \qquad f_{cyc} = 1 / t_{cyc}$$

The registers are introduced behind each function of the suboperation and this creates the pipeline stages. Placing the register at the output (not at the input!!!) makes suboperation stages compatible with the definition of state machines, which are used to control the pipeline.

Data Flow Dependencies

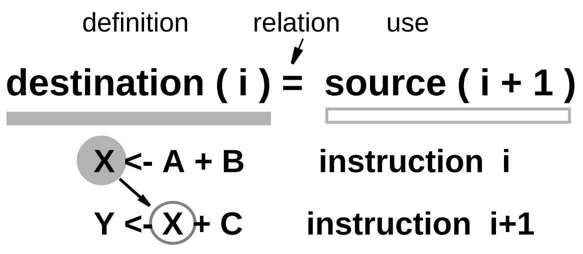
Three different kinds of data flow dependency hazards can occur between two instructions. These data dependencies can be distinguished by the definition-use relation.

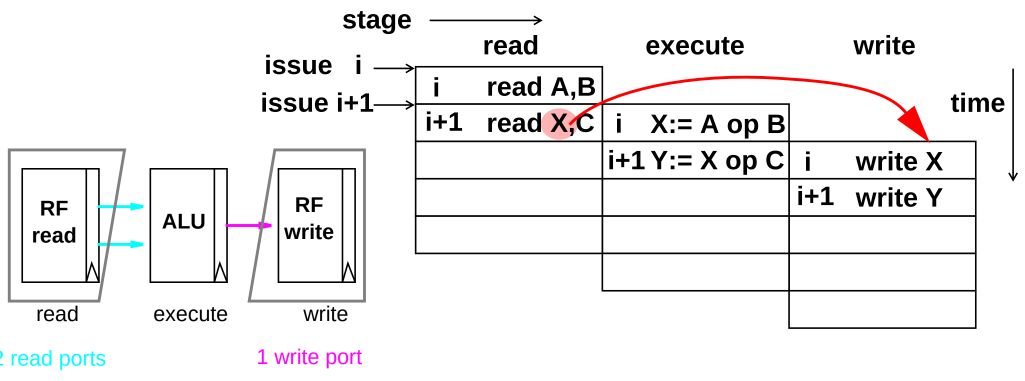
- flow dependency	read after write	RAW
- anti-dependency	write after read	WAR
- output dependency	write after write	WAW

Basically, the only real dependencies in a program are the data flow dependencies, which define the logical sequence of operations transforming a set of input values to a set of output values (see also Data Flow Architectures). Changing the order of instructions must consider these data flow dependencies to keep the semantic of a program.

The **Flow Dependency** read after write (RAW)

This data dependency hazard can occur e.g. in a pipelined processor where values for a second instruction i+1 are needed in the earlier stages and have not yet been computed by the later stages of instruction i. The hazardous condition arises if the dependend instructions are closer together than the number of stages among which the conflicting condition arises.





To avoid the hazard, the two instructions must be separated in time within the pipeline. This can be accomplished by **inserting bubbles** (NOPs) into the pipeline (simply speaken: by waiting for i to complete) or building a special hardware (**hardware interlock**), which inserts NOPs automatically.

Data Flow Dependencies

The **Anti-Dependency** write after read (WAR)

This dependency can only arise in the case of out-of-order instruction issue or out-of-order completion. The write back phase of instruction i+1 may be earlier than the read phase of instruction i. Typical high-performance processors do not use out-of-order issue. This case, then, is of less importance. If the compiler reorders instructions, this reordering must preserve the semantics of the program and take care of such data dependencies.

source (i) = destination (i + 1)

X <- Y + B **instruction i**

Y <- A + C **instruction i+1**

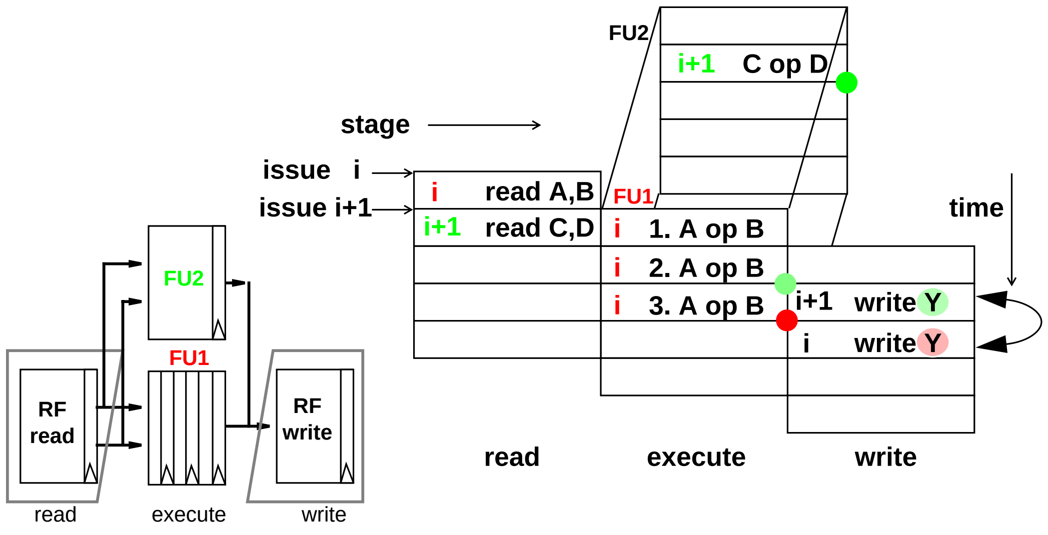
The **Output Dependency** write after write (WAW)

The result y of the first instruction i will be written back to the register file later than y of instruction i+1 because of a longer pipeline for the division. This reverses the sequence of writes i and i+1 and is called out-of-order completion.

destination (i) = destination (i + 1)

Y <- A / B **instruction i**

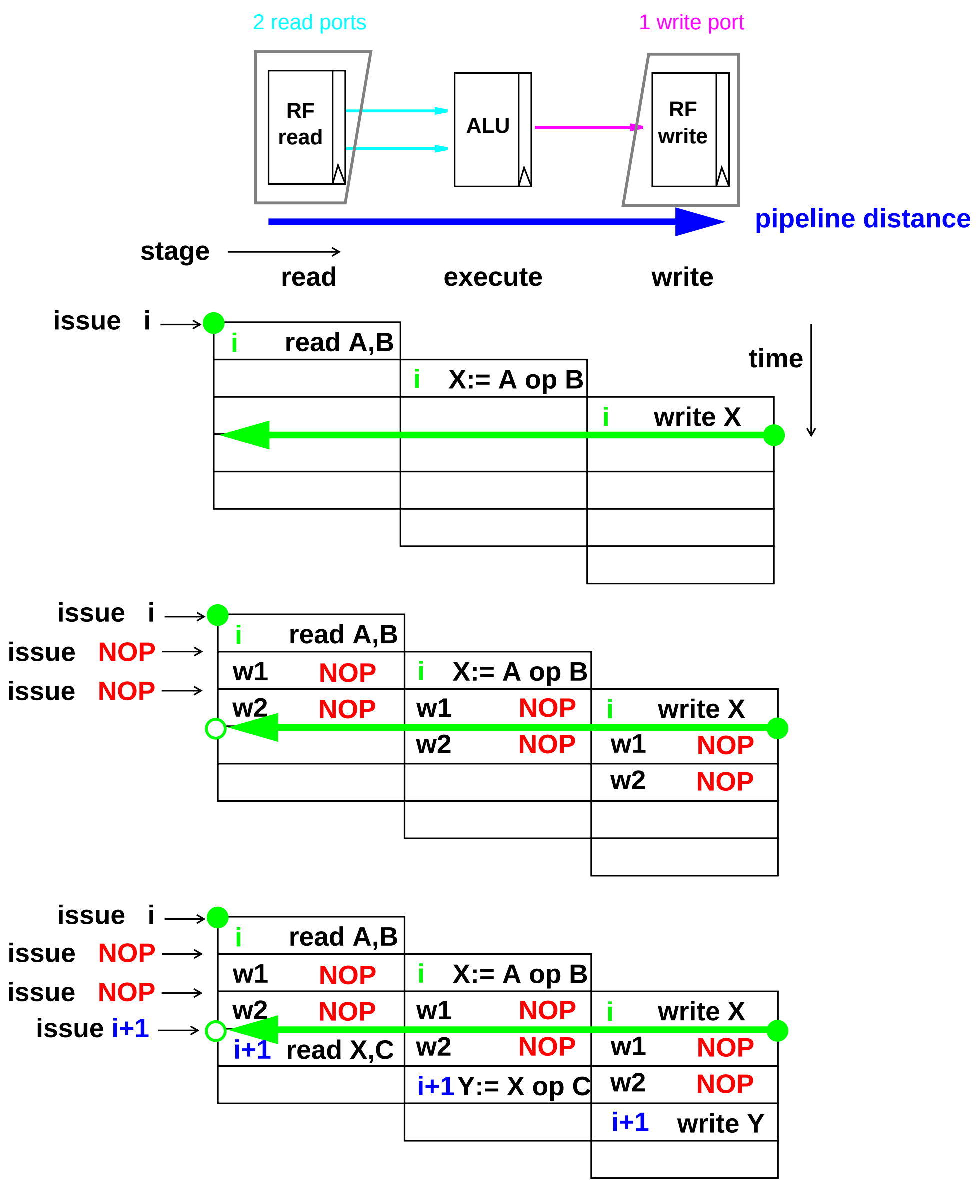
Y <- C + D **instruction i+1**



Data Flow Dependencies

Inserting Bubbles

We must avoid the RAW-hazard in the pipeline, because reading X before it is written back from the previous instruction reads an old value of X and thus destroys the semantic of the two sequential instructions. The sequential execution model (von Neumann architecture) assumes that the previous instruction is completed before the execution advances to the next instruction.



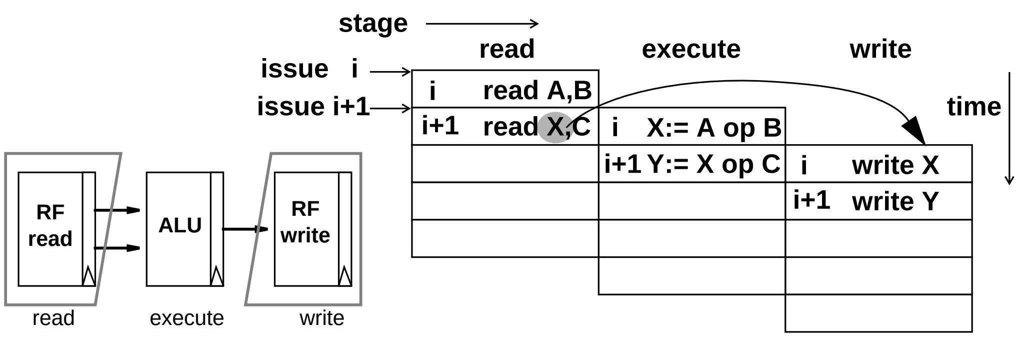
The conflicting stages are RF read and the RF write, which have a distance of 3 pipeline stages. Reading the correct value for X from the register file requires the insertion of 2 NOPs in between the two conflicting instructions. This delays the instruction i+1 by 2 clocks which then removes the RAW-hazard. The compiler (or programmer) is responsible for detecting the hazard and and inserting NOP into the instruction stream.

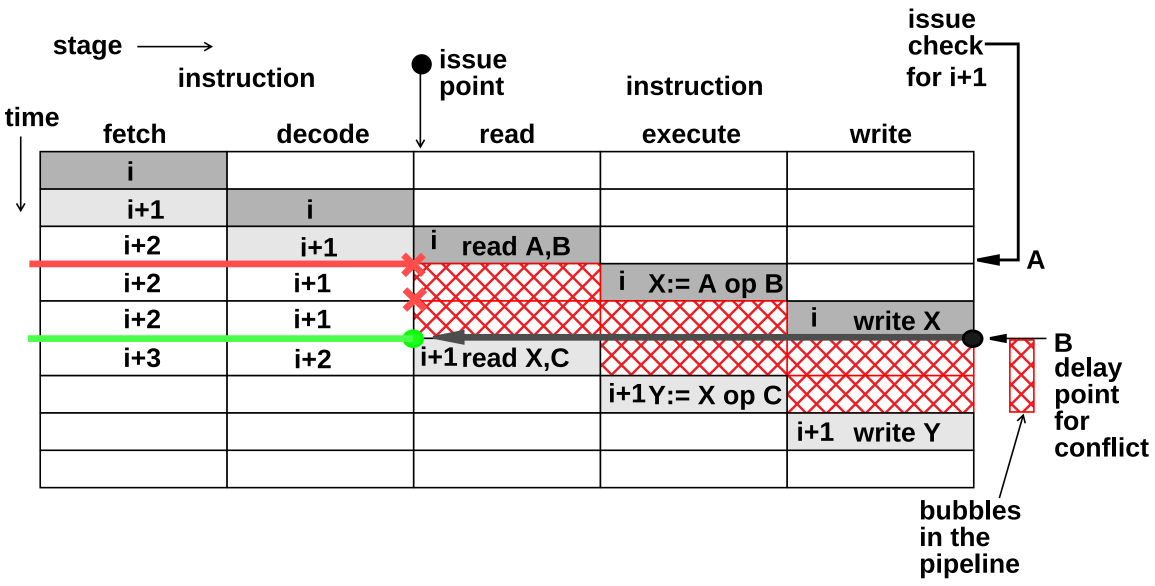
The 2 bubbles in the pipeline can be utilized by other usefull instructions independent from i and i+1 (see instruction scheduling).

Data Flow Dependencies

Hardware Interlock

A hardware mechanism for detecting and avoiding the RAW-Hazard is the interlock hardware. The RAW-Hazard condition is detected by comparing the source fields of instruction i+1 (and i+2) with the destination field of instruction i, until i is written back to the register file and thus the instruction is completely executed.



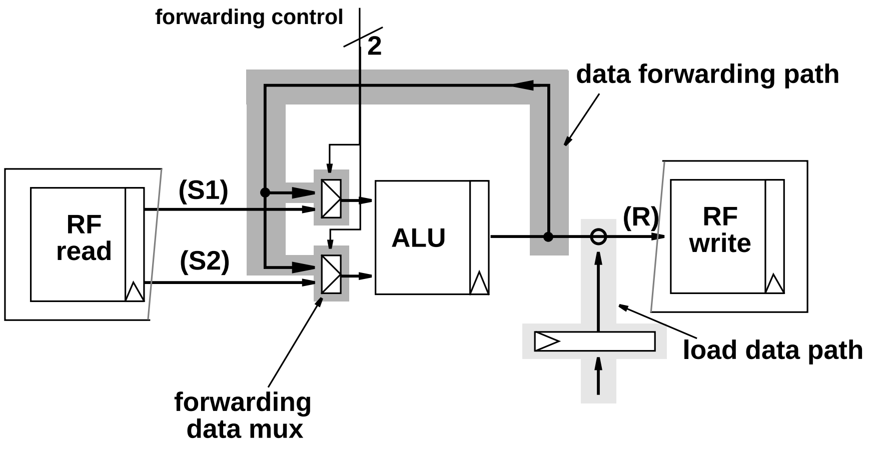


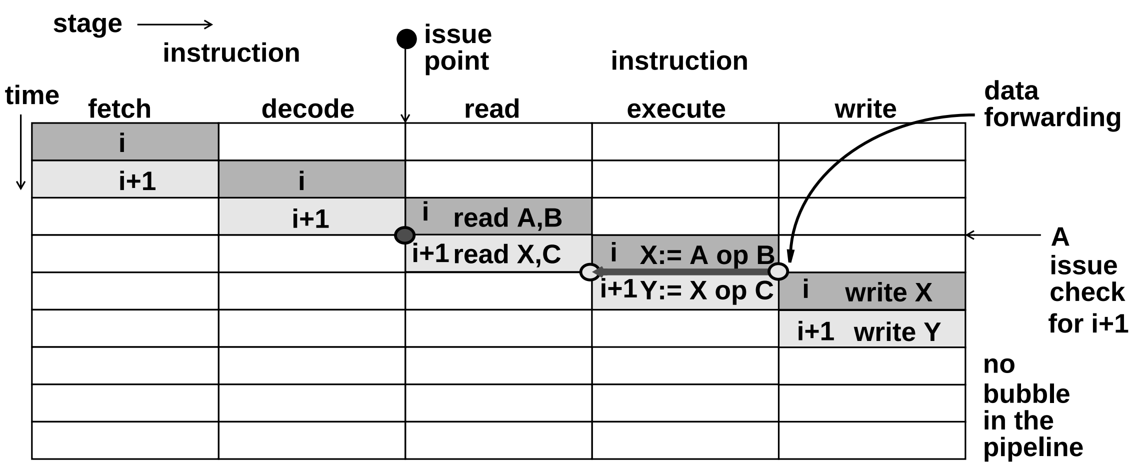
The hardware interlock detects the RAW-Hazard and delays the issue of instruction i+1 as long as the write back of instruction i is completed.

The hardware mechanism doesn't need additional NOPs in the instruction stream. The bubbles are inserted by the hardware.

Nevertheless the produced bubbles can be avoided by scheduling useful and independent instructions in between i and i+1 (see also instruction scheduling).

Data Forwarding





Control Flow Dependencies

The use of pipelining causes difficulties with respect to control-flow dependencies. Every change of control flow potentially interrupts the smooth execution of a pipeline and may produce bubbles in it. One of the following instructions may be responsible for them:

- conditional branch instruction
- jump instruction
- jump to subroutine
- return from subroutine

The bubbles (no operations) in the pipeline reduce the gain of the pipeline thus reducing performance significantly. There are two causes of bubbles. The first one is a data dependency in the pipeline itself, the branch condition being evaluated a pipeline stage later than needed by the instruction fetch stage to determine the correct instruction path. The second one is the latency for accessing the instruction and the new destination for the instruction stream.

Pipeline Utilization

$$U = \frac{n}{n+m} = \frac{1}{1+\frac{m}{n}}$$

n	number of useful instructions
m	number of no operations (bubbles)

$$m = \underbrace{b \bullet (1-p) \bullet N_b}_{\text{no ops caused by branches}} + \underbrace{b \bullet q \bullet N_o}_{\text{no ops caused by the latency of branch target fetches}}$$

b	number of branches
p	probability of correct guesses
N_b	penalty for wrong guess
q	frequency of other causes
N_o	penalty for other causes

$$U = \frac{1}{1 + \frac{b}{n} \bullet (1-p) \bullet N_b + \frac{b}{n} \bullet q \bullet N_o}$$

Control Flow Dependencies

Reduction of Branch Penalty

The ’no’ operations (NOPs) in the pipeline can be reduced by the following techniques:

reduction of branch penalty N_b

- forwarding of condition code
- fast compare
- use of delay slots
- delayed branch
- delayed branch with squashing

increase of p

- static branch prediction
- dynamic branch prediction
- profiled branch prediction

reduction of instruction fetch penalty N_o

- instruction cache
- improvement of instruction prefetch
- branch target buffer
- alternate path following

avoid branches

- predication of instructions

Control Flow Dependencies

Effect of Branches

Before we start to analyse the effects of branching we should introduce the basic terms and have a closer look to the internals of the instruction fetch unit.

Definition : ***Branch successor: The next sequential instruction after a (conditional) branch is named the branch successor.***

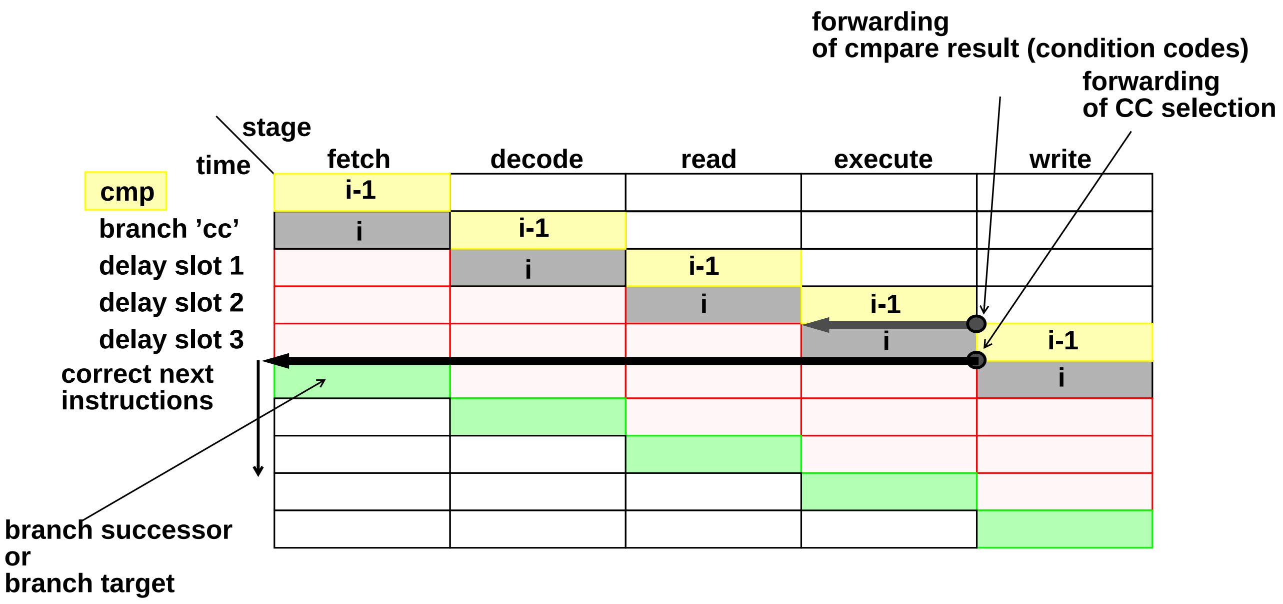
In this case, it is assumed the branch is not taken. The operation of fetching the next instruction in the instruction stream requires to add the distance from the actual instruction to the next instruction to the value of the actual program counter (PC).

Definition : ***Branch target: The instruction to be executed after a branch is taken is called a branch target.***

The operation required to take the branch is to add the branch offset from the instruction word to the PC and then fetch the the branch target instruction from the instruction memory.

The number of pipeline cycles wasted between a branch taken and the resumption of instruction execution of the branch target is called a delay slot. Depending on the pipeline the number of delay slots b may vary between 0 and k-1.

In the following stage-time diagram we assume that the instruction fetch can immediately continue when the branch direction is defined by the selection of the condition code, which is forwarded to the instruction fetch stage.

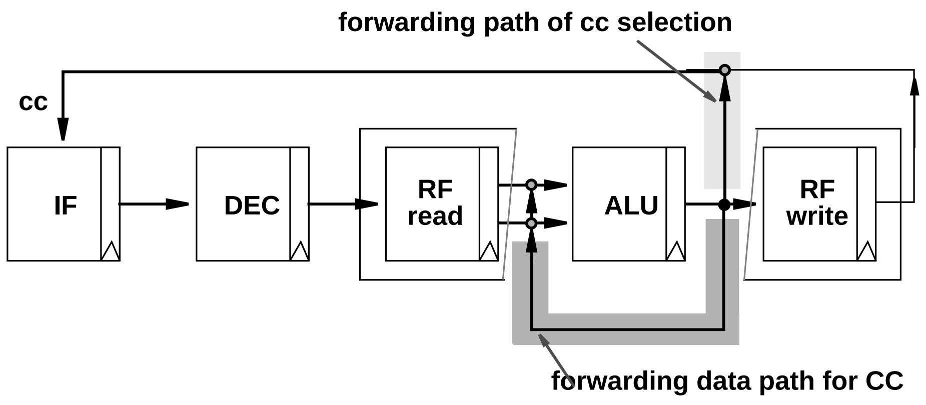


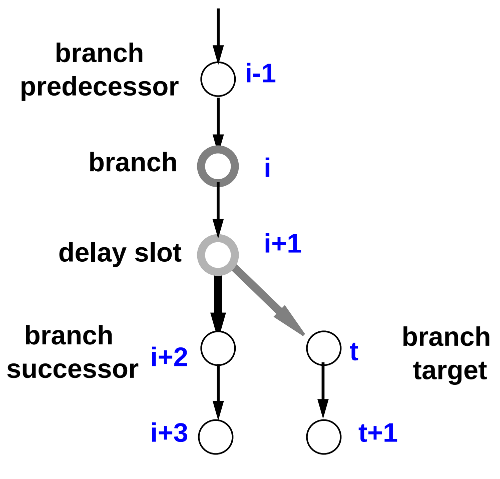
Control Flow Dependencies

Instruction fetch stage

```
cmp R1,R2 -> CC
```
calculate all reasonable bits

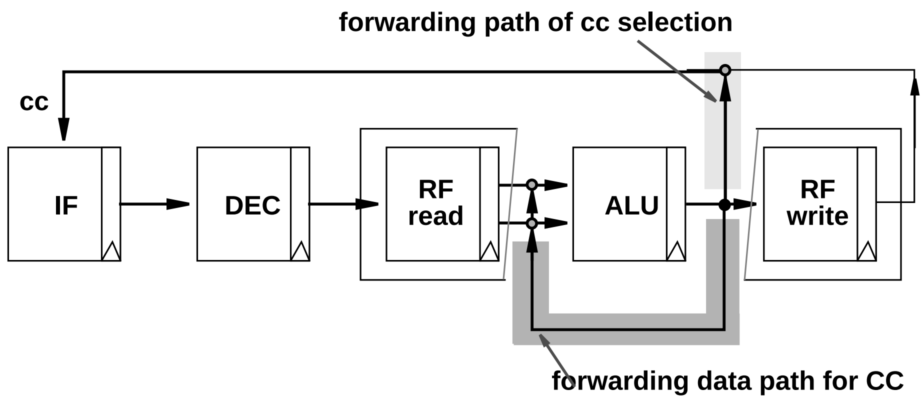
```
bra CC, offset
```
change the control flow
depending on the selection of CC bits
calculate next PC

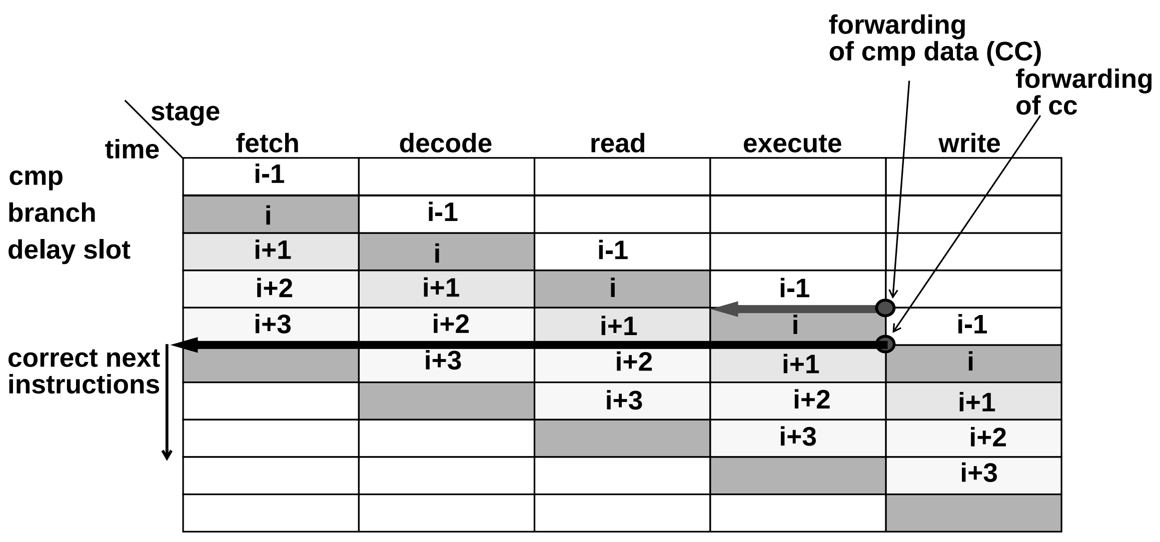




numbering is used to identify the instructions in the flow of processor code

Forwarding of the Condition Code



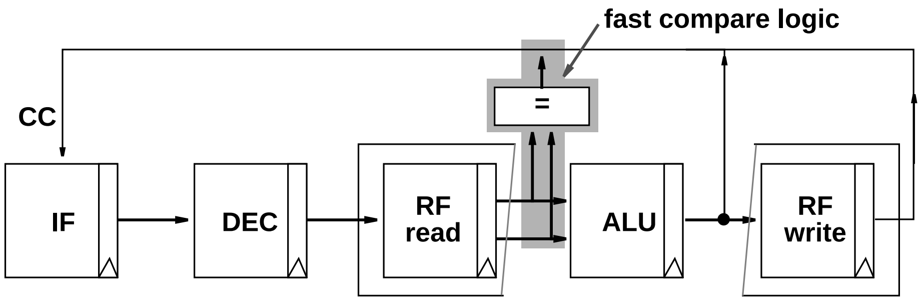

forwarding of cmp data (CC)
forwarding of cc
stage
time
correct next instructions

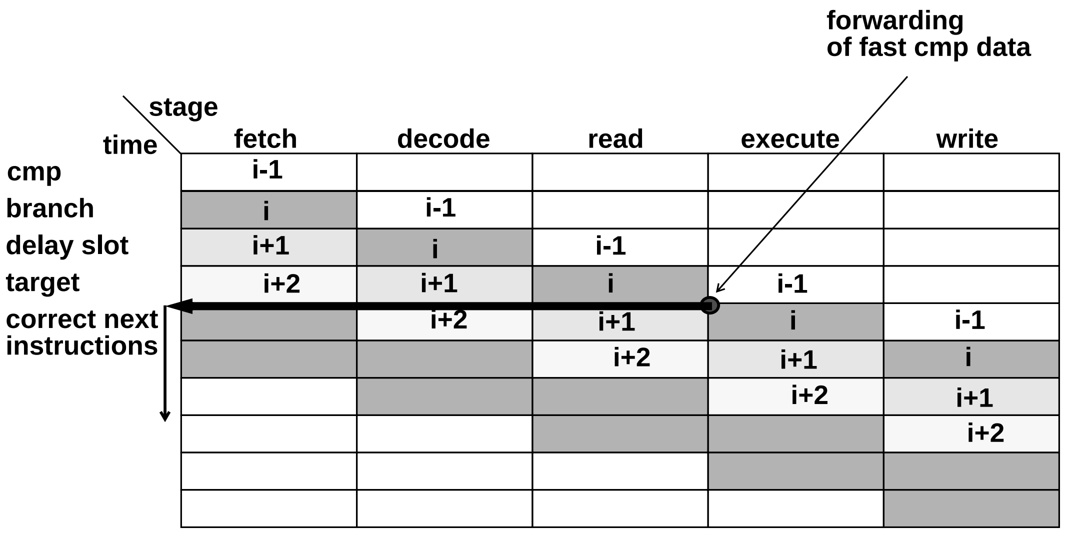
	fetch	decode	read	execute	write
cmp	i-1				
branch	i	i-1			
delay slot	i+1	i	i-1		
	i+2	i+1	i	i-1	
	i+3	i+2	i+1	i	i-1
correct next instructions	▨	i+3	i+2	i+1	i
		▨	i+3	i+2	i+1
			▨	i+3	i+2
				▨	i+3
					▨

Control Flow Dependencies

Fast Compare

simple tests for equal, unequal, <0, <=0, >0, >=0, =0

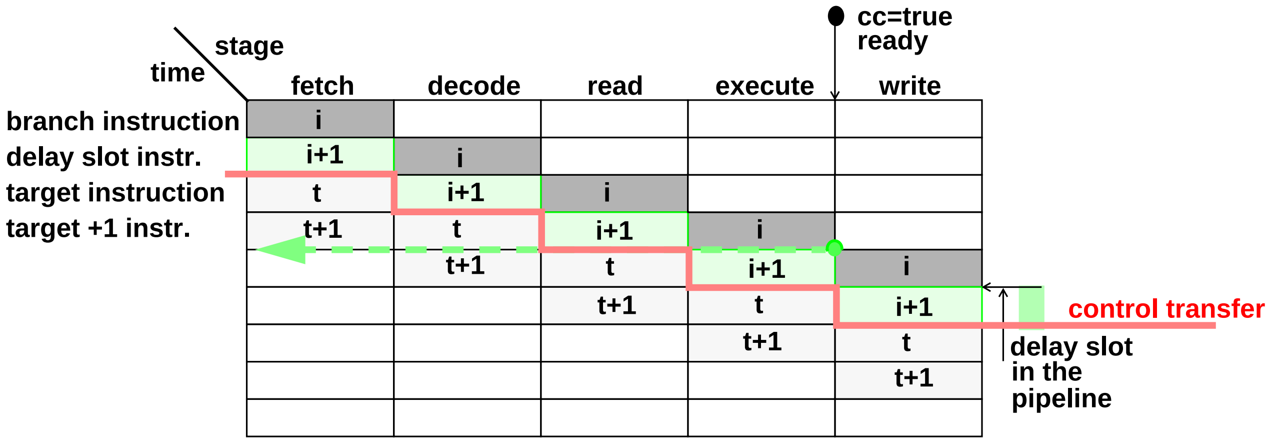




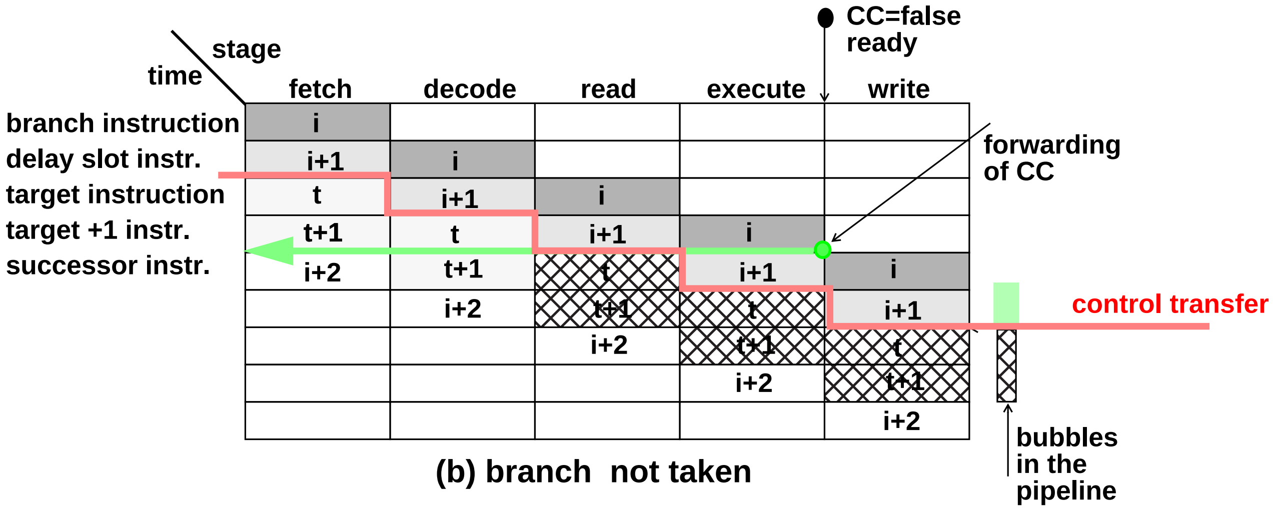
Delayed Branch

Idea: Reducing the branch penalty by allowing <d> useful instructions to be executed before the control transfer takes place.

67% of all branches of a program are taken (loops!). Therefore, it is wise to use the prediction "branch taken".



(a) branch taken



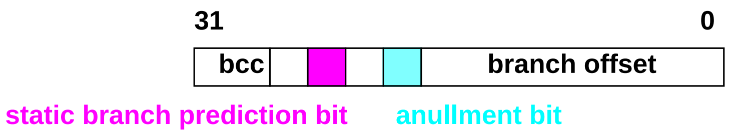
(b) branch not taken

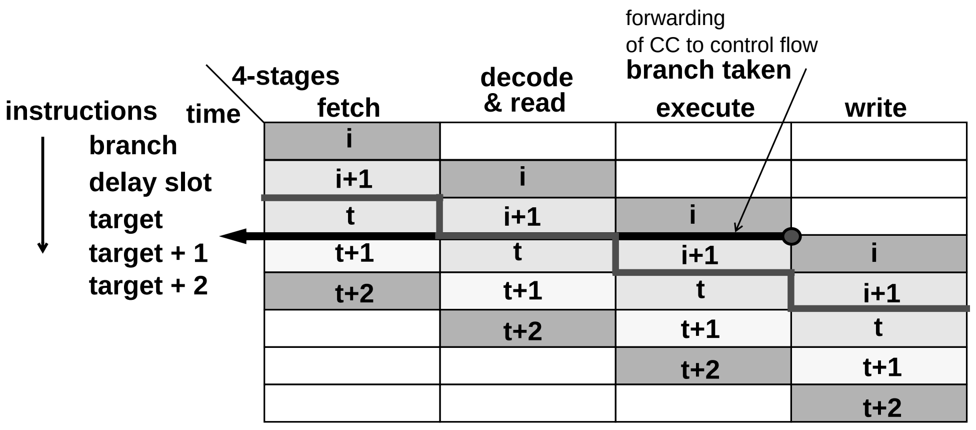
Goal: zero branch penalty

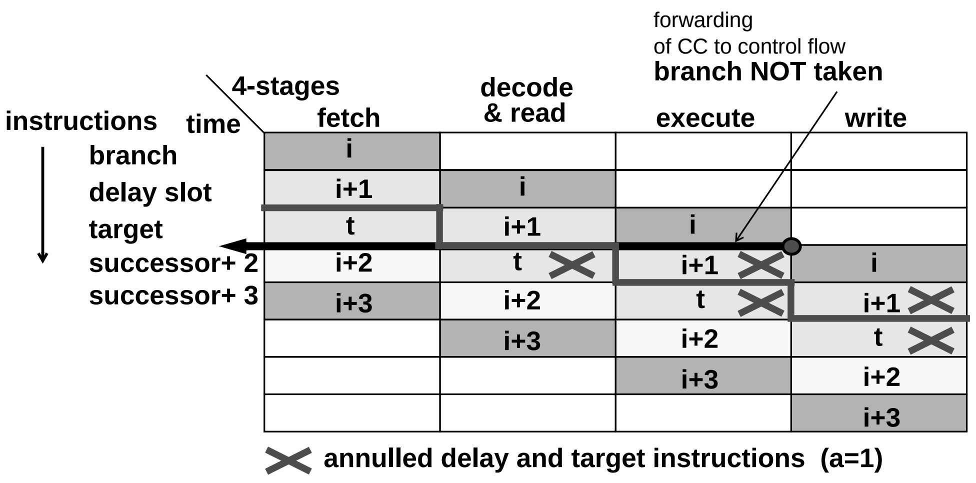
Technique: moving independent instructions into the delay slots.

Probability of using delay slots: 1. slot ~ 0.6; 2. slot ~ 0.2; 3. slot ~ 0.1;

Delayed Branch with Squashing







If the anullment bit is asserted, the delay slot instruction will be anulled, when the branch decision was predicted falsely.

This enables the compiler to schedule instructions from before and after the branch instruction into the delay slot.

Branch Prediction

Two different approaches can be distinguished here:

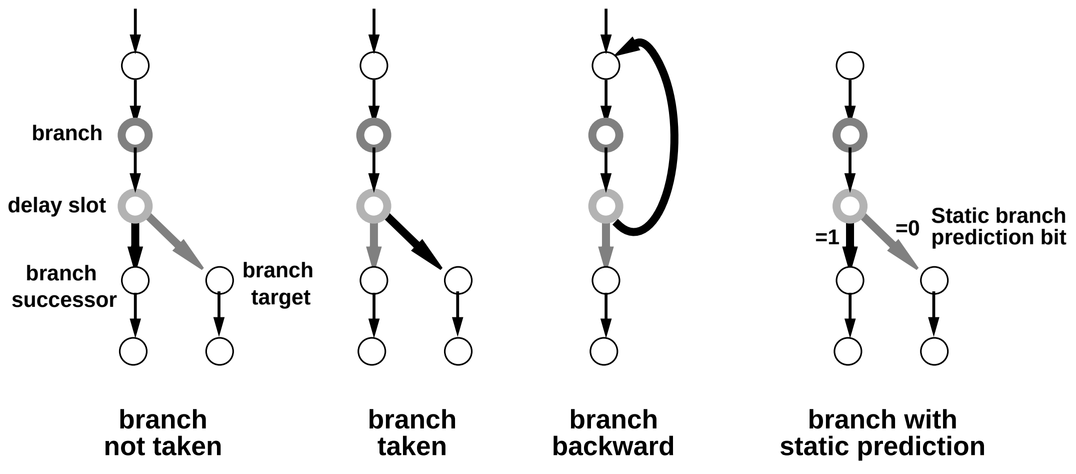
- static-branch prediction
- dynamic-branch prediction

The static-branch information associated with each branch does not change as the program executes. The **static-branch prediction** utilizes compile time knowledge about the behaviour of a branch. The compiler can try to optimize the ordering of instructions for the correct path.

Four prediction schemes can be distinguished:

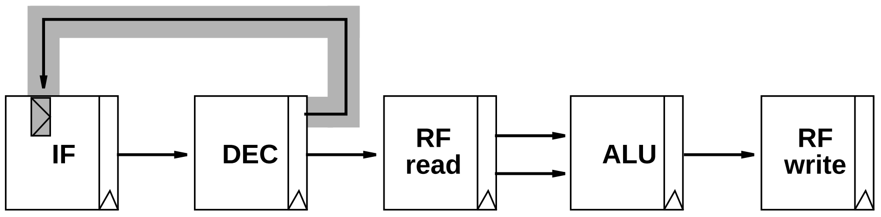
- branch not taken
- branch taken
- backward branch taken
- branch direction bit

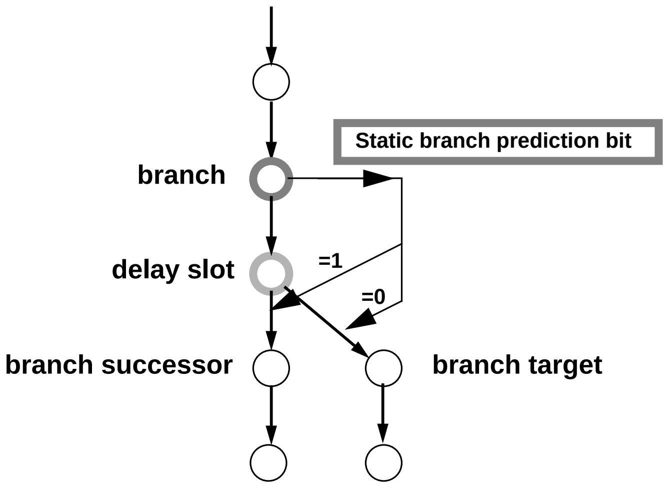
built-in static prediction strategies



Static Branch Prediction

The hardware resources required for static branch prediction are a forwarding path for the prediction bit, and a logic for changing the instruction fetch strategy in the IF stage. This prediction bit supplied by the output of the instruction decode stage decides which path is followed. The prediction bit allows execution of instructions from the branch target or from the successor path directly after the delay slot instruction.





Profiled Branch Prediction

The **profiled branch prediction** is a special case of the static prediction. The guess of the branch direction is based on a test run of the program. This profile information on branch behaviour is used to insert a static branch prediction bit into the branch instruction that has a higher proportion of correct guesses.

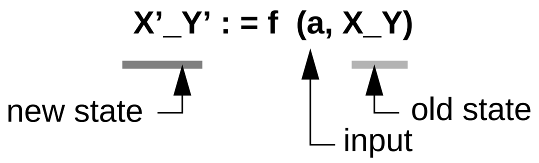
Dynamic Branch Prediction

Dynamic branch prediction uses information on branch behaviour as the program executes. No initial dynamic information exists before the program starts executing.

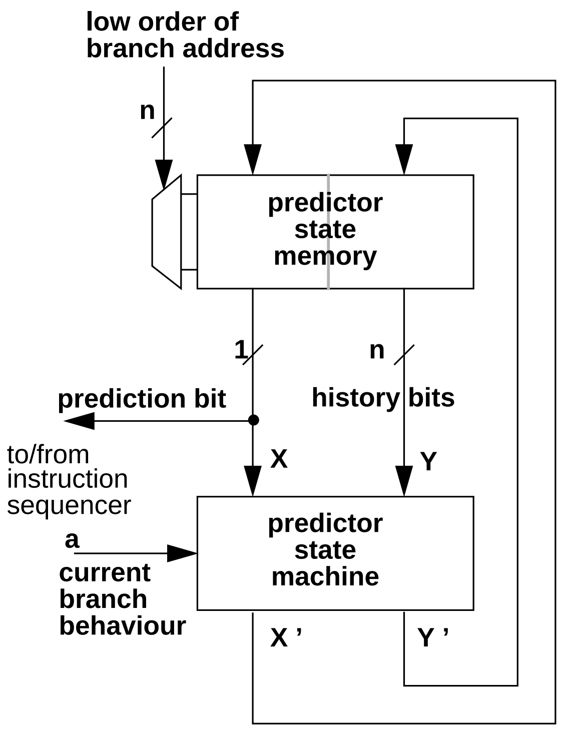
The dynamic information is normally associated with a specific branch. The association can be realized by using the instruction address of the branch as the unique identifier of the branch.

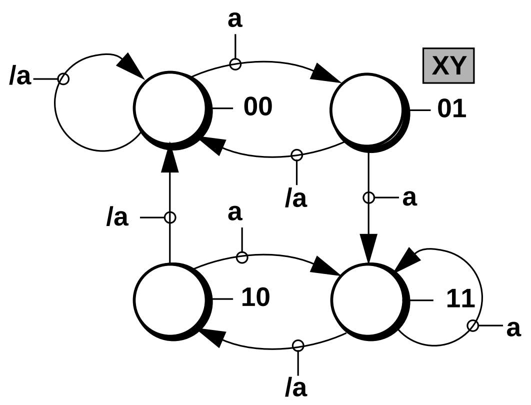
The dynamic prediction depends on the past behaviour of this branch and is stored in a table addressed by the branch instruction address. A unique addressing would need the whole address as an index, usually 32 bit in length. This length prohibits direct use of this address as an index to the state table.

simple branch predictor using only one history bit



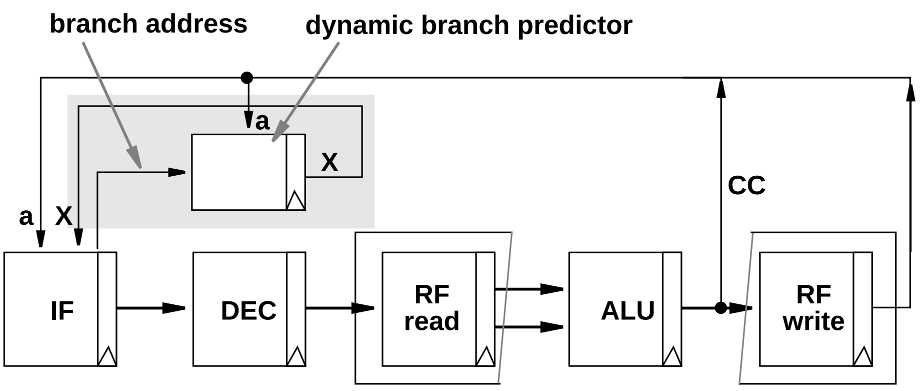
a	current branch behaviour
X	prediction bit
Y	history bits





a	current branch behaviour	
	taken	a
	not taken	/a
X	prediction bit	
	take	1
	take not	0
Y	history bit	
	last branch taken	1
	last branch not taken	0

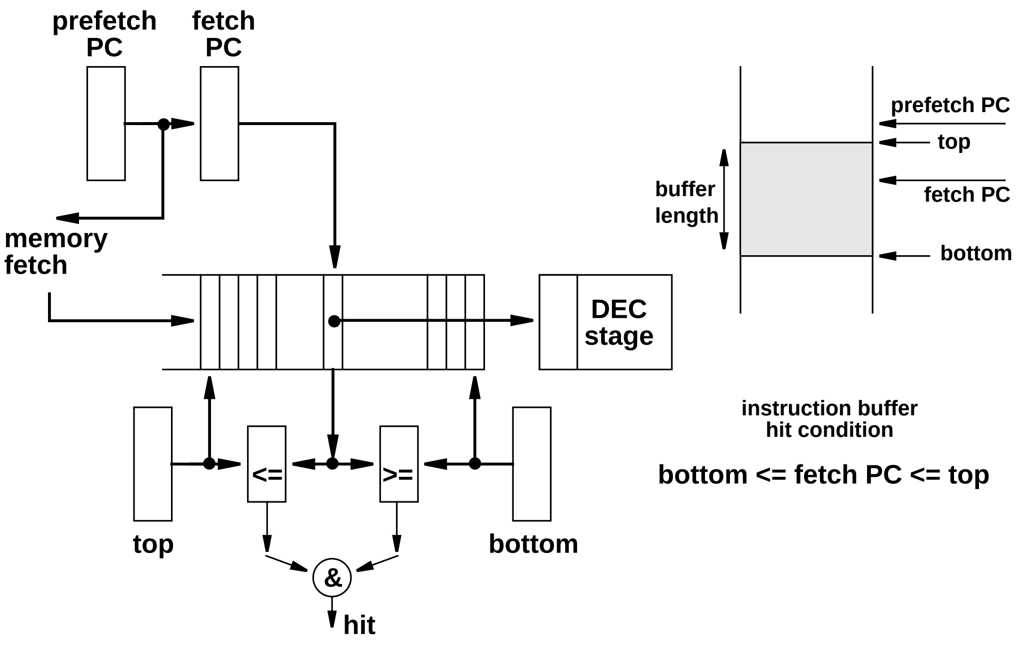
Dynamic Branch Prediction



Reduction of Instruction Fetch Penalty

instruction cache

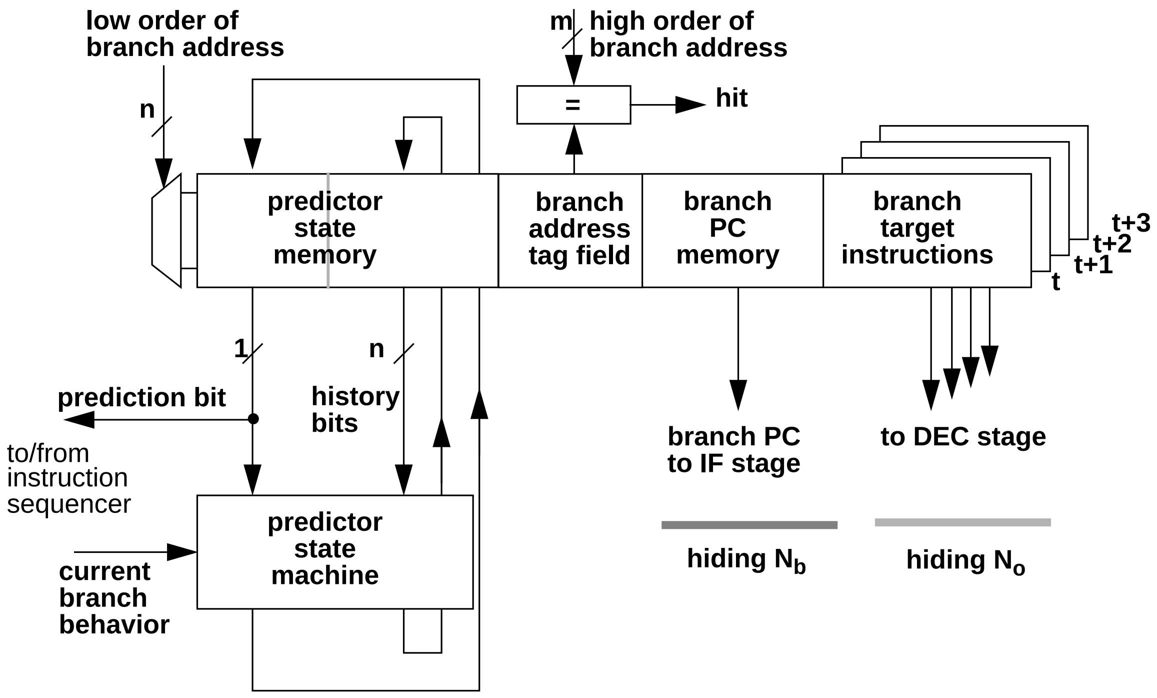
instruction buffer



Hardware Resources for an Instruction Buffer

Reduction of Instruction Fetch Penalty

Branch Target Buffer


low order of branch address
n
m
high order of branch address
=
hit
predictor state memory
branch address tag field
branch PC memory
branch target instructions
t+3
t+2
t+1
t
1
n
prediction bit
history bits
to/from instruction sequencer
predictor state machine
current branch behavior
branch PC to IF stage
to DEC stage
hiding N_b
hiding N_o