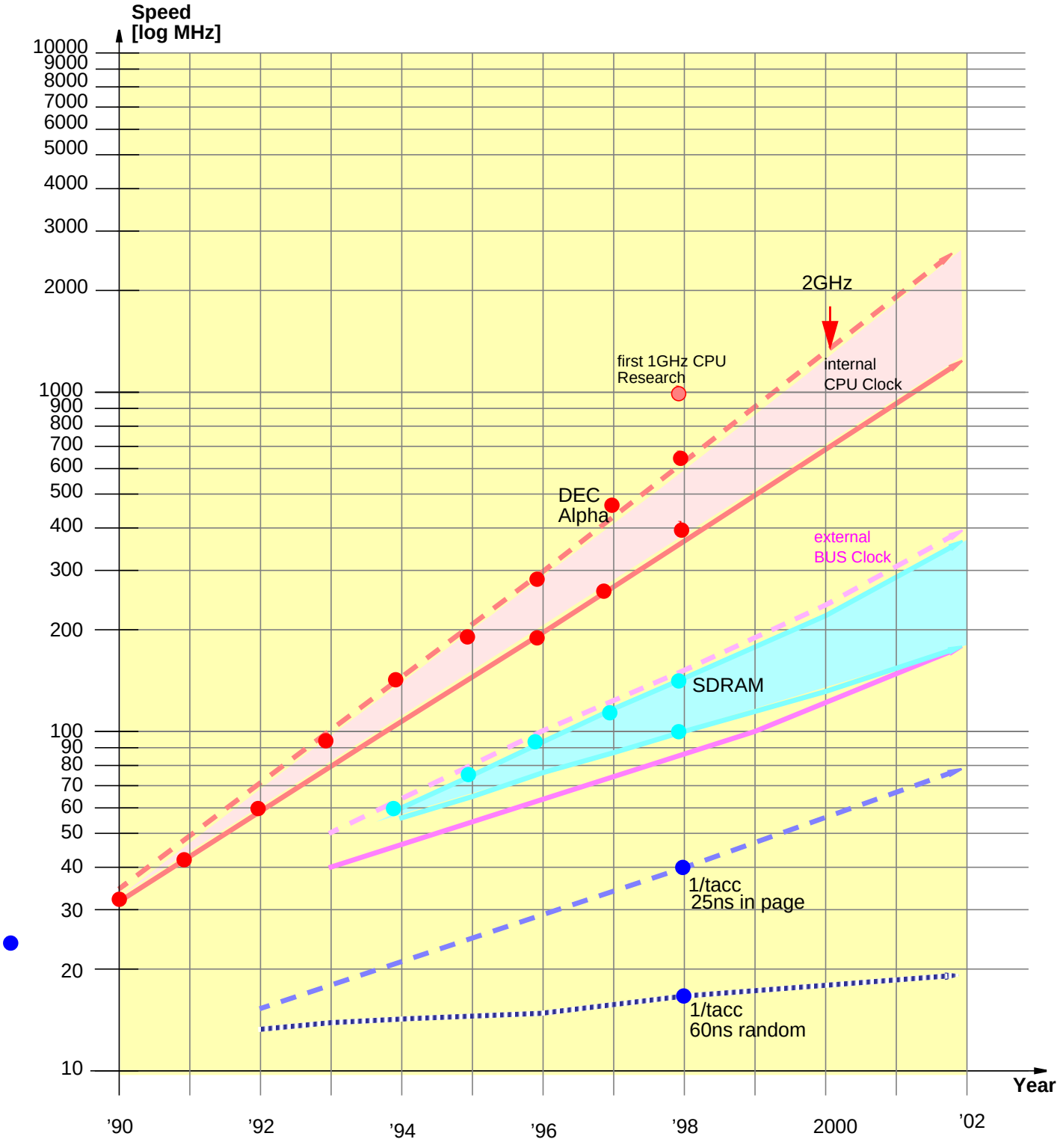
DRAMs

Speed Trends: DRAMs & Processors

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.

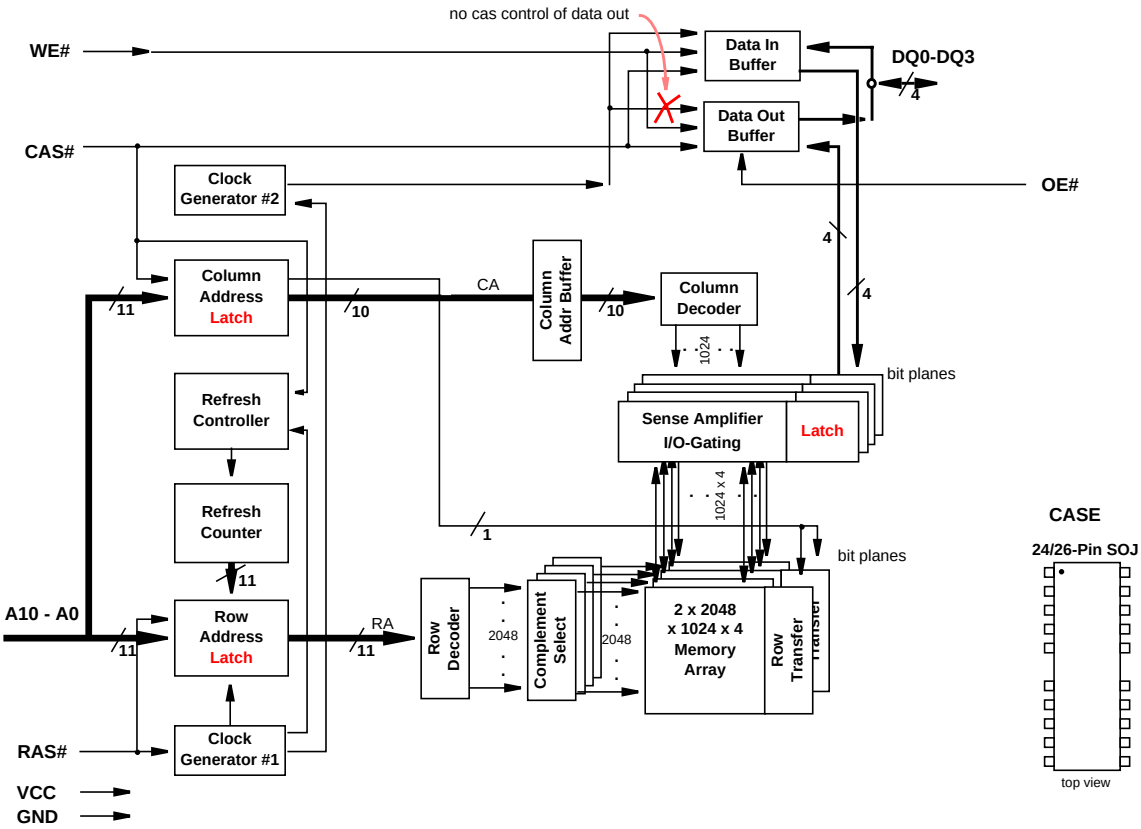


EDO-DRAM

DRAM: Beispiel EDO~

Extended Data-Out (EDO) DRAMs feature a fast Page Mode access cycle. The EDO-DRAM separates three-state control of the data bus from the column address strobe, so address sequencing-in is independent from data output enable.

This Page Access allows faster data operation within a row-address-defined page boundary. The PAGE cycle is always initiated with a row-address strobed-in by the folling edge of RAS_ followed by a column-address strobed-in by the folling edge of CAS_. CAS_ may be toggled by holding RAS_ low and ’strobing-in’ different column-addresses, thus executing faster memory cycles.



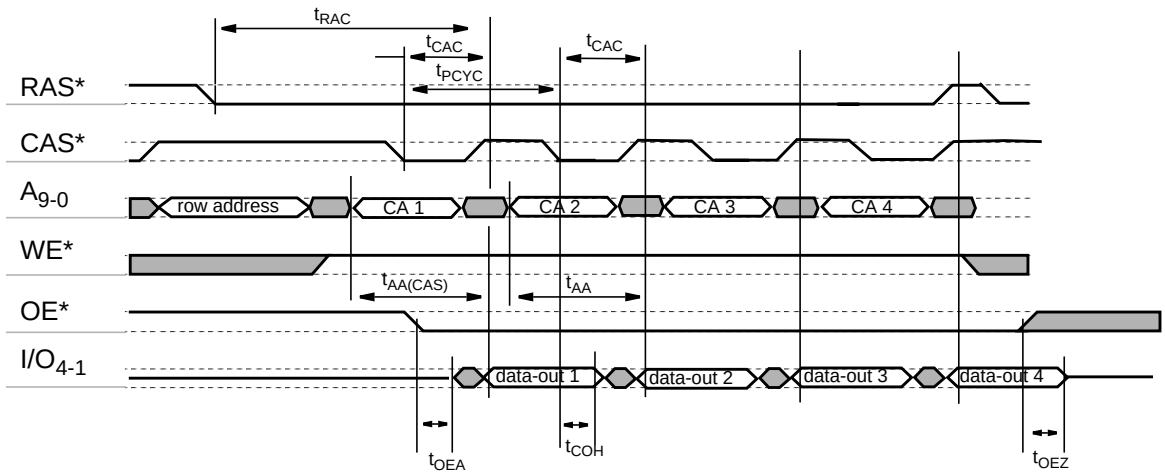
EDO Operation: Improvement in page mode cycle time was achieved by converting the normal sequential fast page mode operation into a two-stage pipeline. A page address is presented to the EDO-Dram, and the data at that selected address is amplified and latched at the data output drivers. While the output buffers are driving the data off-chip, the address decode and data path circuitry is reset and able to initiate access to the next page address.

EDO-DRAM

Timing EDO-DRAM

The primary advantage of EDO is the availability of data-out even after /CAS_ high. EDO allows CAS-Precharge time to occur without the output data going invalid. This elemination of CAS Output control allows "pipelining" of READs. [Micron, Data Sheet MT4LC4ME8(L), 1995.

The term pipelining is misused in this context, it is simply an overlapping of addressing and data-transfer.

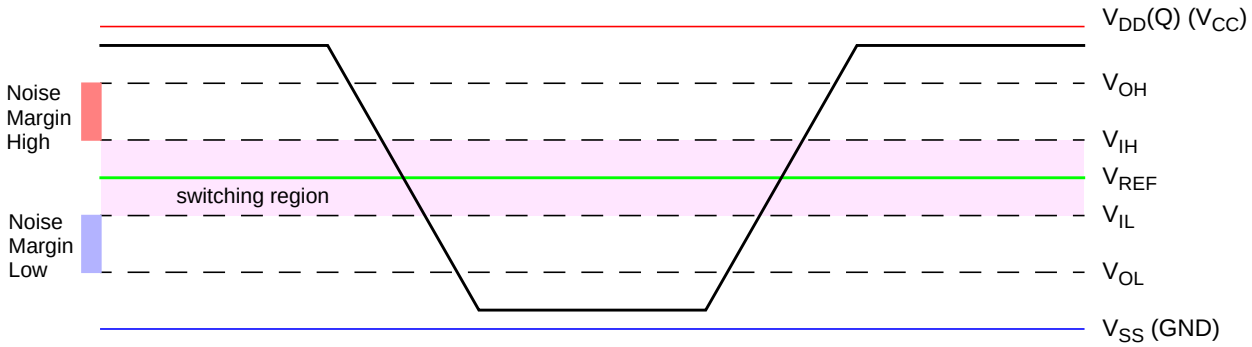


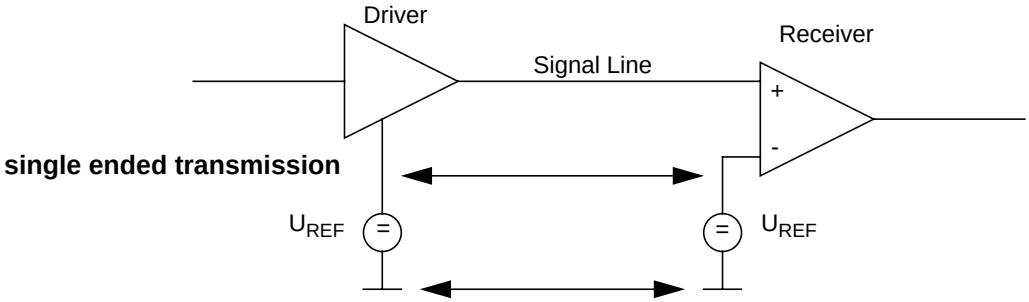
hyperpage mode cycle EDO

Im Gegensatz zum ’fast page mode’-DRAM muß CAS_ nicht low sein, um den Datenausgangs-Buffer zu treiben. Die nächste CA kann somit unabhängig von der Data-Output Phase früher in das CA-Latch übernommen werden. Das Treiben der Daten wird ausschließlich vom Signal OE_ gesteuert (t_{OEA} und t_{OEZ}). Die Daten am Ausgang schalten erst auf die neuen Daten von der nächsten CA um, wenn die neue CA im DRAM wirksam wird (nach t_{COH}).

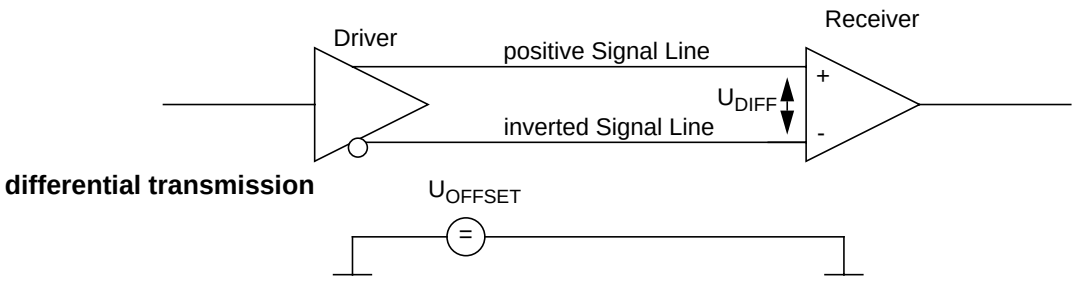
Memory Interface Signaling

High speed interfaces





	LVTTL	CTT	BTL	GTL	HSTL	SSTL
VOH	2,4	1,9	2,1	1,2	1,1	Vtt + 0,4
VIH	2	1,7	0,85	0,85	1,85	Vtt + 0,2
VREF	1,5	1,5	1,55	0,8	0,75	1,5 = Vtt
VIL	0,8	1,3	1,47	0,75	0,65	Vtt - 0,2
VOL	0,4	1,1	1,1	0,4	0,4	Vtt - 0,4
VDD (Q)	3,3	3,3	5	1,2	1,5	3,3
NM(H)	0,4	0,2	0,48	0,35	0,25	0,2
NM(L)	0,4	0,2	0,37	0,35	0,25	0,2
SWING(O)	2	0,8	1	0,8	0,7	0,8



Memory Management

Ziele des Memory Management

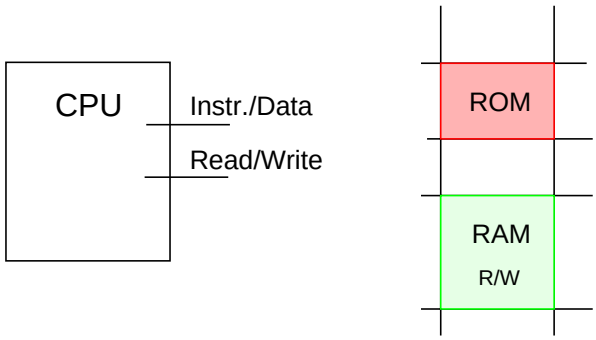
- Schutzfunktion => Zugriffsrechte
- Speicherverwaltung für Prozesse
- Erweiterung des begrenzten physikalischen Hauptspeichers

Eine sehr einfache Möglichkeit den Speicher zu organisieren ist die Aufteilung in einen Festwertspeicher (Read-Only-Memory) für das Programm und in einen Schreib-Lese-Speicher für die Daten.

- R/W Memory

- ROM - kann nicht überschrieben werden

Eine solche feste Aufteilung ist nur für '**single tasking**'-Anwendungen sinnvoll, wie sie z.B. in 'eingebetteten Systemen' mit Mikrocontrollern verwendet werden.



Literature:
Andrew S. Tanenbaum, "Structured Computer Organization", 4th edition, Prentice-Hall, p. 403ff.

Beim Multi-processing oder **Multi-tasking** Systemen reicht die oben genannte Möglichkeit, den Speicher zu organisieren, nicht aus. Es existieren viele Prozesse, die quasi gleichzeitig bearbeitet werden müssen.

Probleme:

- Verlagern von Objekten im Speicher - relocation
- Schutz von Objekten im Speicher - protection

Lösung:

Einführung eines logischen Adreßraumes pro Prozeß und einer Abbildung der logischen Adreßräume auf den physikalischen Adreßraum (Hauptspeicher), die Adreßumsetzung - 'address translation'

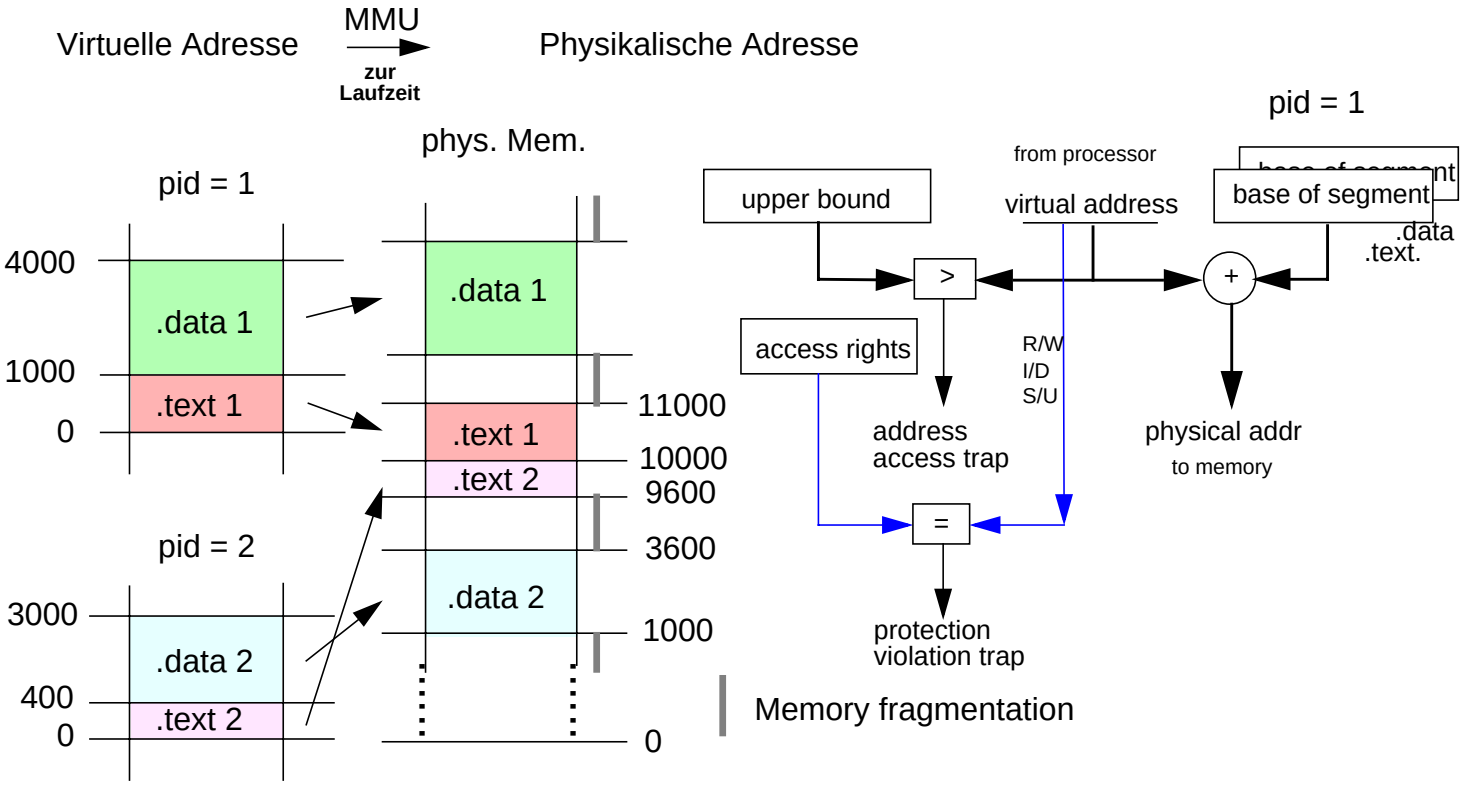
=> viele Prozesse konkurrieren um den physikalischen Speicher!

=> um ausgeführt zu werden, müssen alle Segmente (.text / .data / .bss) in den Speicher geladen werden

Segmentierung ist die Aufteilung des logischen Adreßraumes in kontinuierliche Bereiche unterschiedlicher Größe zur Speicherung von z.B. Instruktionen (.text) oder Daten (.data) oder Stack-Daten (.bss), etc.

Jedes Segment kann jetzt mit Zugriffsrechten geschützt werden.

Memory Management



Um einen Prozeß ausführen zu können, müssen alle Segmente des Prozesses im Hauptspeicher sein.

- Der Platz ist bereits durch andere Prozesse belegt

  => Es müssen so viele Segmente anderer (ruhender) Prozesse aus dem Speicher auf die Platte ausgelagert werden, wie der neue auszuführende Prozess Platz benötigt. Randbedingung durch die Segmentierung: es muß ein fortlaufender Speicherbereich sein!

  - Swapping: Aus- und Einlagerung ganzer Prozeßsegmente (Cache flush!)

- Es ist zwar noch Platz vorhanden, aber die Lücken reichen nicht für die benötigten neuen Segmente aus. Wiederholtes Aus- und Einlagern führt zu einer Fragmentierung des Speichers -'Memory fragmentation'

  => Es müssen Segmente verschoben werden, d.h. im Speicher kopiert werden, um Lükken zu schließen (Cache flush!).

  In den gängigen Architekturen wird mit n=32 oder 64 Bit adressiert. Daraus folgt die Größe des virtuellen Adressraums mit 2^n Byte (2^{32}= 4 GByte; 2^{64}= ?? TBytes)

- Der Hauptspeicherplatz reicht von seiner Größe nicht für den neuen Prozeß aus

  => der Hauptspeicher wird durch das Konzept des **Virtuellen Speichers** durch die Einbeziehung des Sekundärspeichers (Platte) erweitert.

Memory Management

Literatur:

Hwang, Kai; Advanced Computer Architecture, Mc Graw Hill, 1993.

Stone, Harold S.; High Performance Computer Architecture, Addison Wesley, 1993

Giloi: Rechnerarchitektur

Tannenbaum, Andrew S.: Modern Operating Systems, Prentice-Hall, 1992.

Intel: i860XP Microprocessor Programmers Reference Manual

Motorola: PowerPC 601 RISC Microprocessor User's Manual

● Grundlagen

Die Memory Managment Unit (MMU) ist eine Hardwareeinheit, die die Betriebssoftware eines Rechners bei der Verwaltung des Hauptspeichers für die verschiedenen auszuführenden Prozesse unterstützt.

- address space protection
- virtual memory - demand paging
- segmentation

Each word/byte in the **physical memory (PM)** is identified by a unique physical address. All memory words in the main memory forms a **physical address space (PAS)**.

All program-generated (or by a software process generated) addresses are called **virtual addresses (VA)** and form the **virtual address space (VAS)**.

When **address translation** is enabled, the MMU maps instructions and data virtual addresses into physical addresses before referencing memory. Address translation maps a set of vitual addresses V uniquely to a set of physical addresses M.

Virtual memory systems attempt to make optimum use of main memory, while using an auxiliary memory (disk) for backup. VM tries to keep active items in the main memory and as items became inactive, migrate them back to the lower speed disk. If the management algorithms are successful, the performance will tend to be close to the performance of the higher-speed main memory and the cost of the system tend to be close to the cost per bit of the lower-speed memory (optimization of the memory hierarchy between main memory and disk).

Most virtual memory systems use a technique called **paging**. Here, the physical address space is divided up into equallly sized units called **page frames.** A **page** is a collection a data that occupies a page frame, when that data is present in memory. The pages and the page frames are always of the same fixed size (e.g. 4K Bytes).

Memory Management

● Virtueller Speicher / Paging

Logischer und physikalischer Adressraum werden in Seiten fester Größe unterteilt, meist 4 oder 8KByte. Logische Pages werden in einer Pagetable auf physikalische Pageframes abgebildet, dabei ist der logische Adressraum im allgemeinen wesentlich größer als der physikalisch vorhandene Speicher. Nur ein Teil der Pages ist tatsächlich im Hauptspeicher, alle anderen sind auf einen Sekundärspeicher (Platte) ausgelagert.

- Programme können größer als der Hauptspeicher sein
- Programme können an beliebige physikalischen Adressen geladen werden, unabhängig von der Aufteilung des physikalischen Speichers
- einfache Verwaltung in Hardware durch feste Größe der Seiten
- für jede Seite können Zugriffsrechte (read/write, User/Supervisor) festgelegt und bei Zugriffen überprüft werden
- durch den virtuellen Speicher wird ein kostengünstiger großer und hinreichend schneller Hauptspeicher vorgetäuscht (ähnlich Cache)

Die Pagetable enthält für jeden Eintrag einen Vermerk, ob die Seite im Hauptspeicher vorhanden ist (P-Bit / present). Ausgelagerte Pages müssen bei einer Referenz in den Hauptspeicher geladen werden, ggf. wird dabei eine andere Page verdrängt. Modifizierte Seiten (M-Bit / modify) müssen dabei auf den Sekundärspeicher zurückgeschrieben werden. Dazu wird ein weiteres Bit eingeführt, das bei einem Zugriff auf die Seite gesetzt wird (R-Bit / referenced)

Die Abbildung des virtuellen Adressraums auf den physikalischen erfolgt beim paging durch die Festlegung von Zuordnungspaaren (VA-PA).

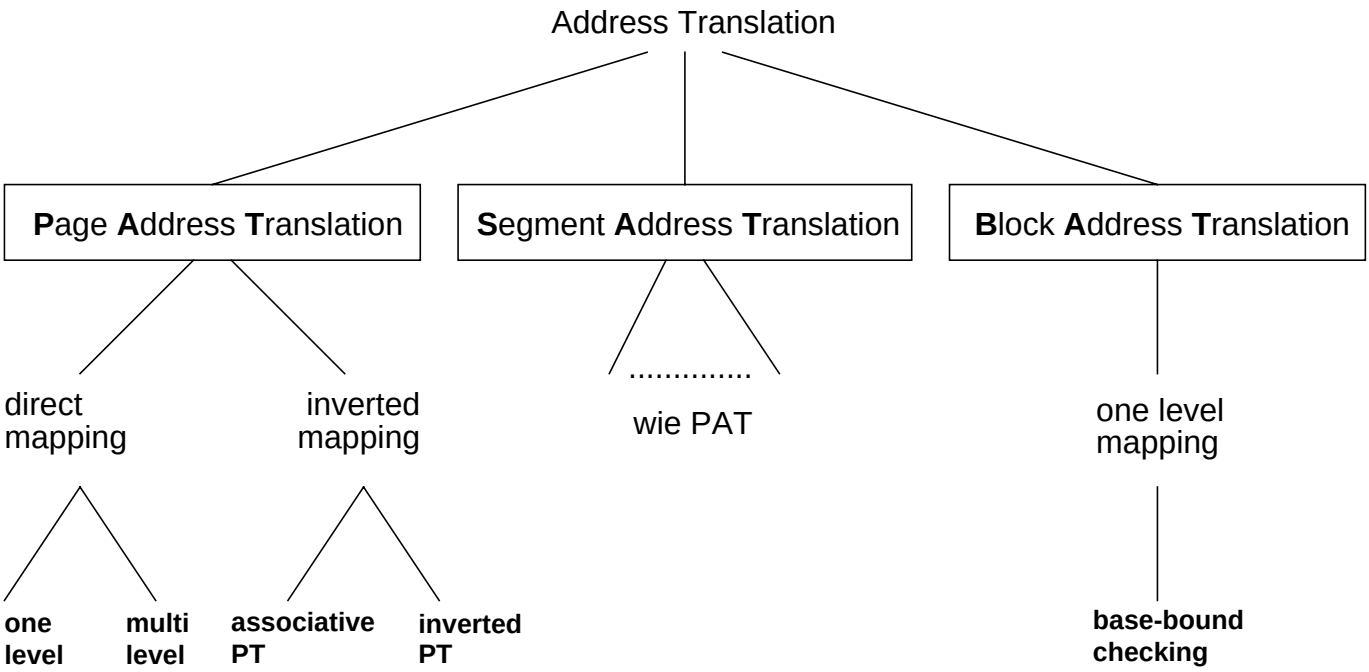
Virtueller Speicher / Paging

Replacement-Strategien :

- not recently used - NRU
  mithilfe der Bits R und M werden vier Klassen von Pages gebildet
  0: not referenced, not modified
  1: not referenced, modified (empty class in single processor systems!)
  2: referenced, not modified
  3: referenced, modified
  es wird eine beliebige Seite aus der niedrigsten nichtleeren Klasse entfernt
- FIFO
  die älteste Seite wird entfernt (möglicherweise die am häufigsten benutzte)
- Second-Chance / Clock
  wie FIFO, wurde der älteste Eintrag benutzt, wird zuerst das R-Bit gelöscht und die nächste Seite untersucht, erst wenn alle Seiten erfolglos getestet wurden, wird der älteste Eintrag tatsächlich entfernt
- least recently used - LRU
  die am längsten nicht genutzte Seite wird entfernt, erfordert Alterungsmechnismus

Memory Management

Verfahren zur Adreßtransformation



PAT: Es wird eine Abbildung von VA nach PA vorgenommen, wobei eine feste Page Size vorausgesetzt wird. Der Offset innerhalb einer Page ist damit eine feste Anzahl von bits (**l**ast **s**ignificant **b**it (LSB)) aus der VA, die direkt in die PA übernommen werden.

Der Offset wird also nicht verändert !

Die Abbildung der höherwertigen Adreßbits erfolgt nach den oben genannten Mapping-Verfahren.

BAT: Provides a way to map ranges of VA larger than a single page into contiguous area of physical memory (typically no paging)

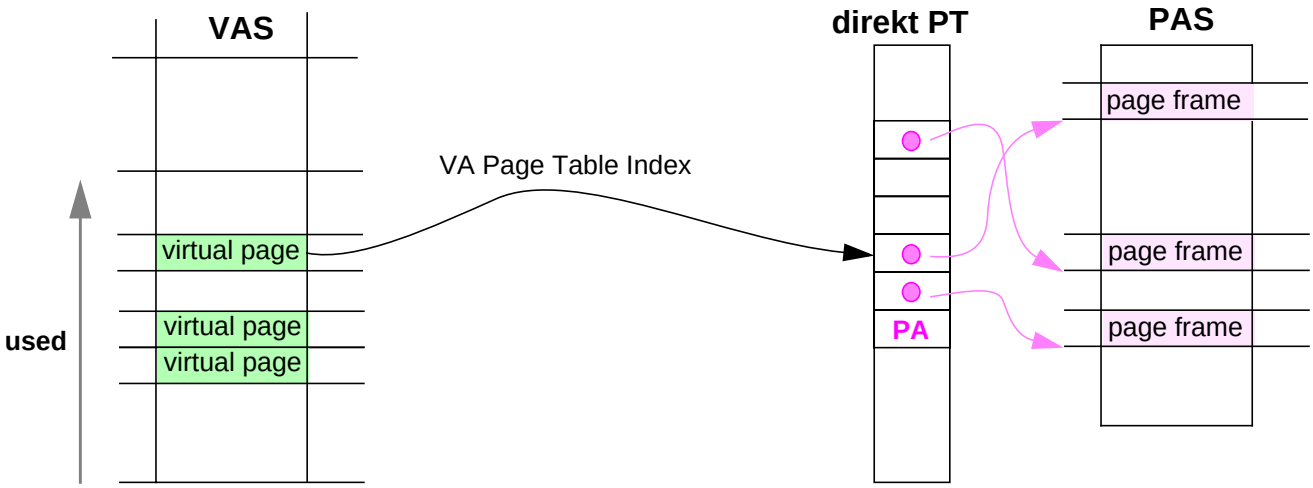
- Used for memory mapped display buffers or large arrays of MMU data.
- base-bound mapping scheme
- block sizes 128 KB (2^{17}) to 256 MB (2^{28})
- fully associative BAT registers on chip
  small number of BAT entries (4 ... 8)

+ BAT entries have priority over PATs

Memory Management

Direct Page Table

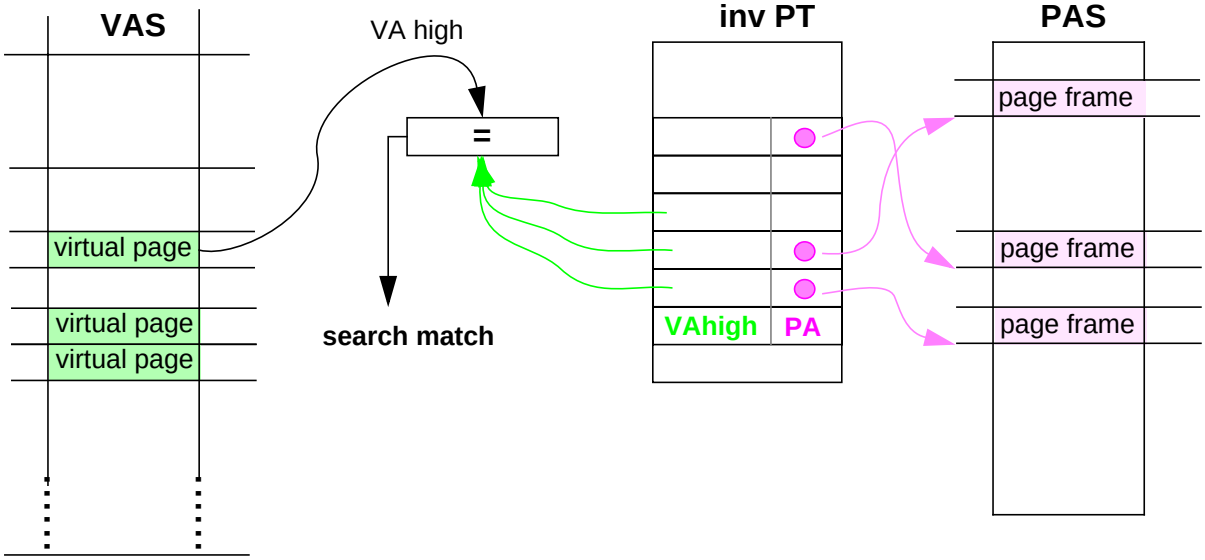
For all used VA-Frames exist one PT-Entry which maps VA to PA. There are much more entries required as physical memory is available.



Inverted Page Table

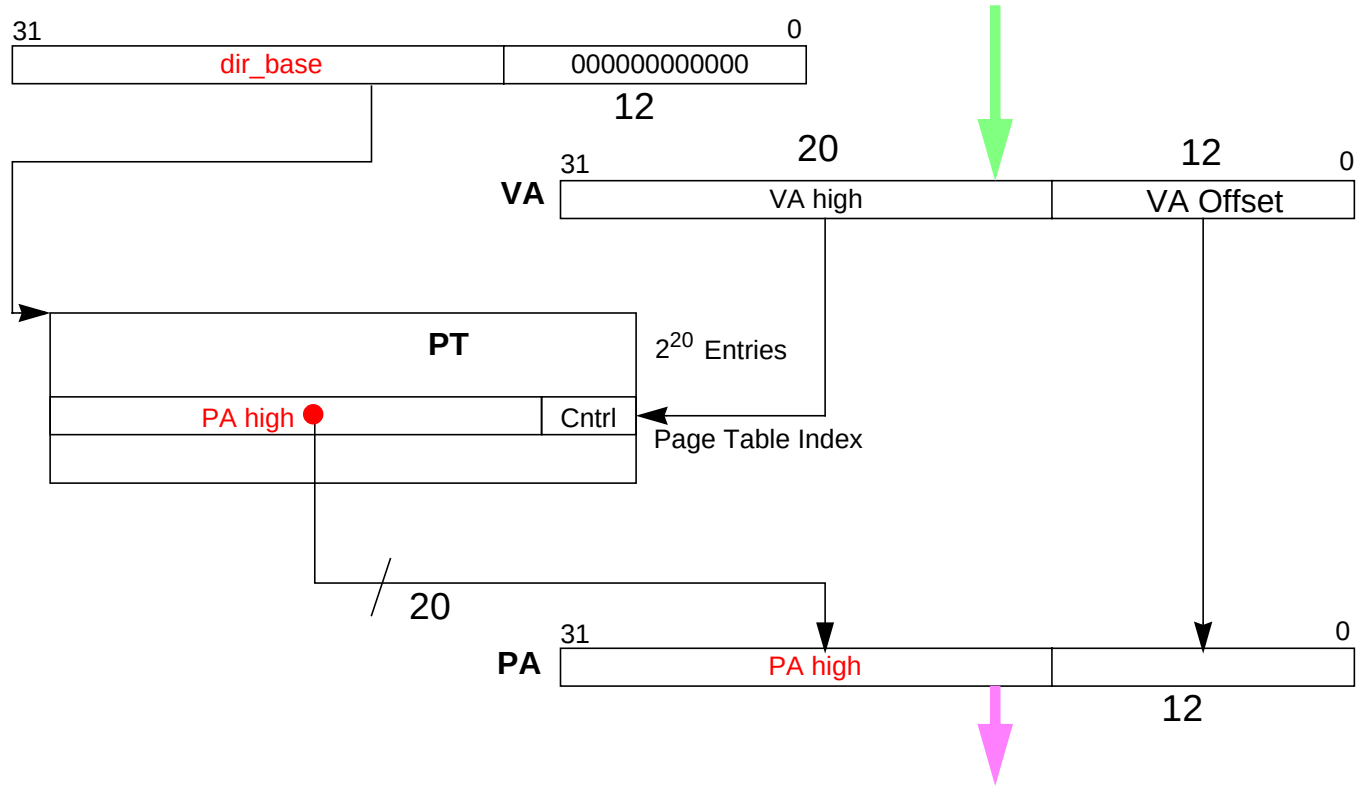
For all **PA** exist one PT-Entry. Indexing with the VA doesn't work any more!!!

To find the right mapping entry a search with the VA as key must be performed by linear search or by associative match. Used for large **v**irtual **a**ddress **s**pace VAS, e.g. 2^{64} and to keep the PT small.



Memory Management

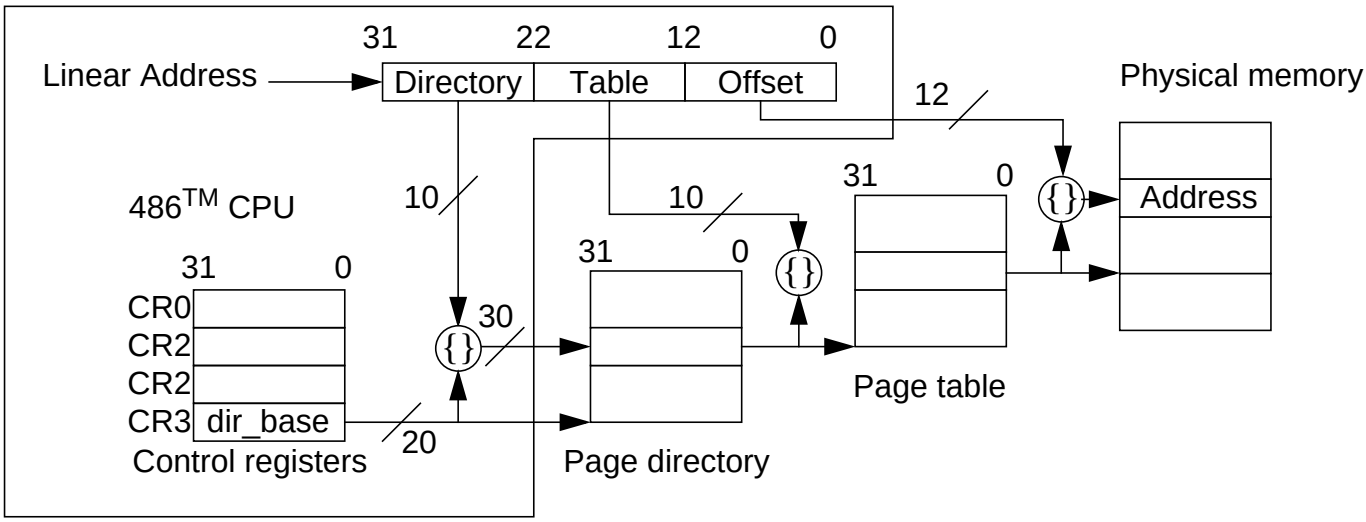
Einstufiges Paging



Die virtuelle Adresse VA wird in einem Schritt in eine physikalische Adresse PA umgesetzt. Dazu wird ein höherwertiger Teil der VA zur Indizierung in die Page Table PT verwendet. In der PT findet man dann unter jedem Index genau einen Eintrag mit dem höherwertigen Teil der PA. Diese einstufige Abbildung kann nur für kleine Teile der VA high verwendet werden (Tabellengröße 4 MB bei 32 bit entries)

Memory Management

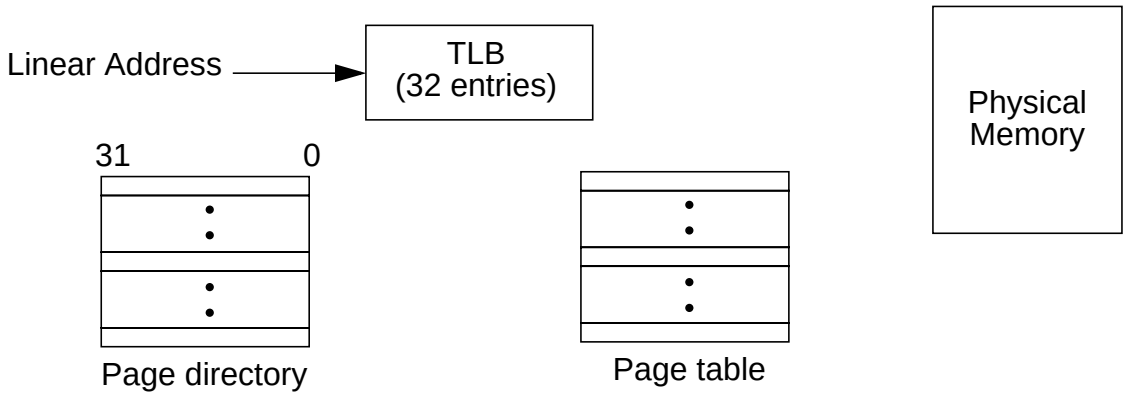
Mehrstufiges Paging



Bei 32 Bit Prozessoren und einer Seitengröße von z.B. 4 KByte wird die Pagetable sehr groß, z.B. 4 MByte bei 32 Bit Pagetable Einträgen. Da meist nicht alle Einträge einer Tabelle wirklich genutzt werden, wird eine mehrstufige Umsetzung eingeführt. Zum Beispiel referenzieren die obersten Adressbits eine Tabelle, während die mittleren Bits den Eintrag in dieser Tabelle selektieren.

- einzelne Tabellen werden kleiner und in der zweiten Stufe werden nur wenige Tabellen benötig
- die Tabellen der zweiten Ebene können selbst ausgelagert werden

Pagetables können aufgrund ihrer Größe nur im Hauptspeicher gehalten werden. Pagetables sind prinzipiell cachable, allerdings werden die Einträge wegen ihrer relativ seltenen Benutzung (im Vergleich zu normalen Daten) schnell aus dem allgemeinen Cache verdrängt.



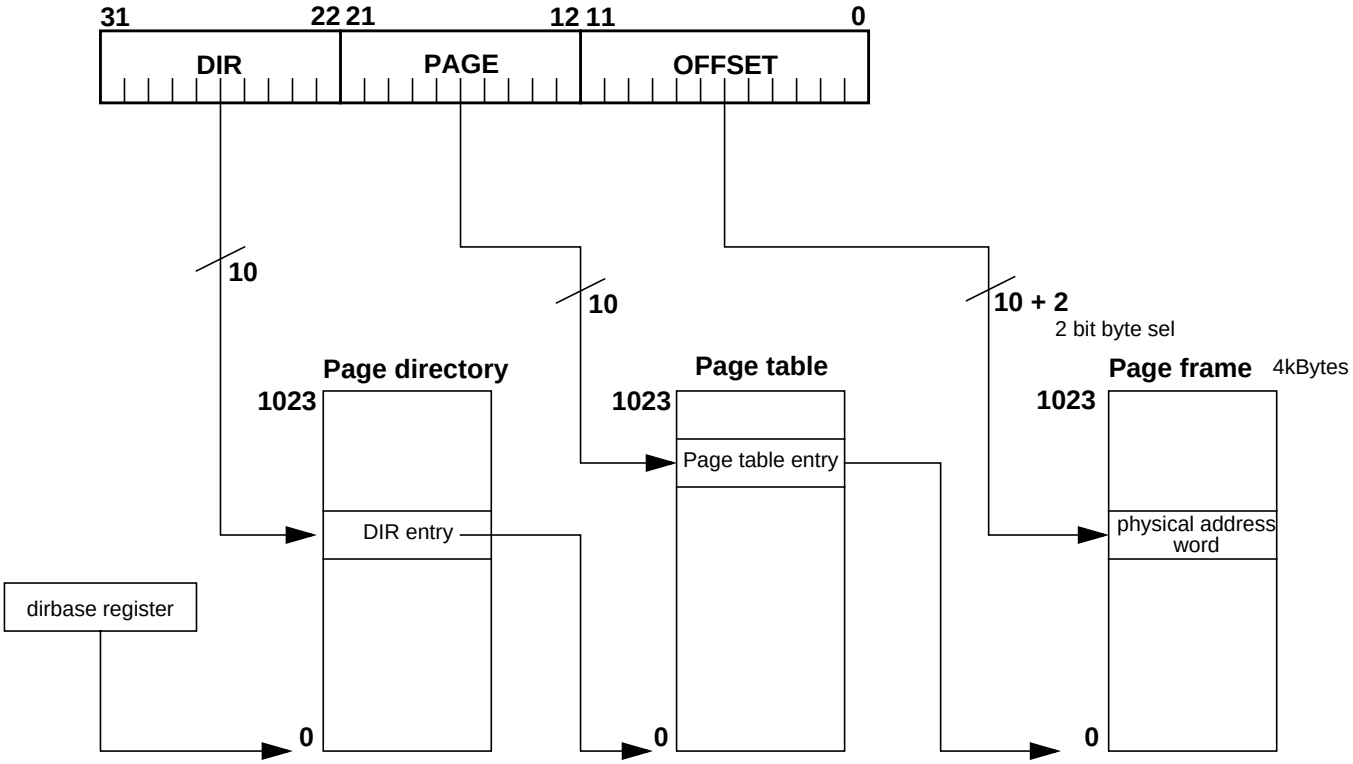
Zum Beschleunigen der Adressumsetzung, insbesondere bei mehrstufigen Tabellen, wird ein Cache verwendet. Dieser Translation Lookaside Buffer (TLB) oder auch Address Translation Cache (ATC) enthält die zuletzt erfolgten Adressumsetzungen. Er ist meist vollassoziativ ausgeführt und enthält z.B. 64 Einträge. Neuerdings wird auch noch ein setassoziativer L2-Cache davorgeschaltet.

Memory Management

● i860XP

A virtual address refers indirectly to a physical address by specifying a page table through a directory page, a page within that table, and an offset within that page.

Format of a Virtual Address



two-level Page Address Translation

A page table is simply an array of 32-bit page specifiers. A page table is itself a page (1024 entries with 4 bytes = 4kBytes). Two levels of tables are used to address a page frame in main memory. Page tables can occupy a significant part of memory space (2^{10} x 2^{10} words = 2^{22} bytes; 4MBytes).

The physical address of the current page directory is stored in the DTB (Directory table base) field of the dirbase register.

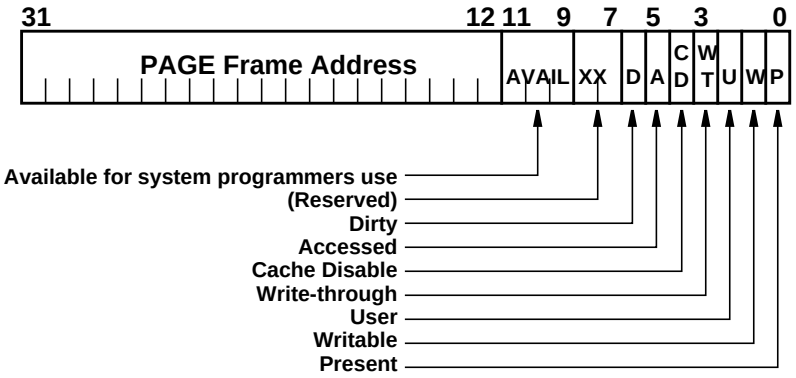
Memory Management

● i860XP

A page table entry contains the page frame address and a number of management bits.

The present bit can be used to implement demand paging. If P=0, than that page is not present in main memory. An access to this page generates a trap to the operating system, which has to fetch the page from disk, set the P bit to 1 and restart the instruction.

Format of a Page Table Entry (i860XP)



Definition : ***Page***
The virtual address space is divided up into equal sized units called pages. [Tanenbaum]
Eine Page ist die Unterteilung des virtuellen und physikalischen Speichers in gleich große Teile.

Definition : ***Page frame***
Ist der Speicherbereich im Hauptspeicher in den genau eine Page hineinpaßt.

Memory Management

Hashing

Literatur: Sedgewick, Robert, Algorithmen, Addison-Wesley, 1991, pp.273-287

Hashing ist ein Verfahren zum Suchen von Datensätzen in Tabellen

- Kompromiß zwischen Zeit- und Platzbedarf

Hashing erfolgt in zwei Schritten:

1. Berechnung der Hash-Funktion

   Transformiert den Suchschlüssel (key, hier die VA) in einen Tabellenindex. Dabei wird der Index wesentlich kürzer als der Suchschlüssel und damit die erforderliche Tabelle kleiner.

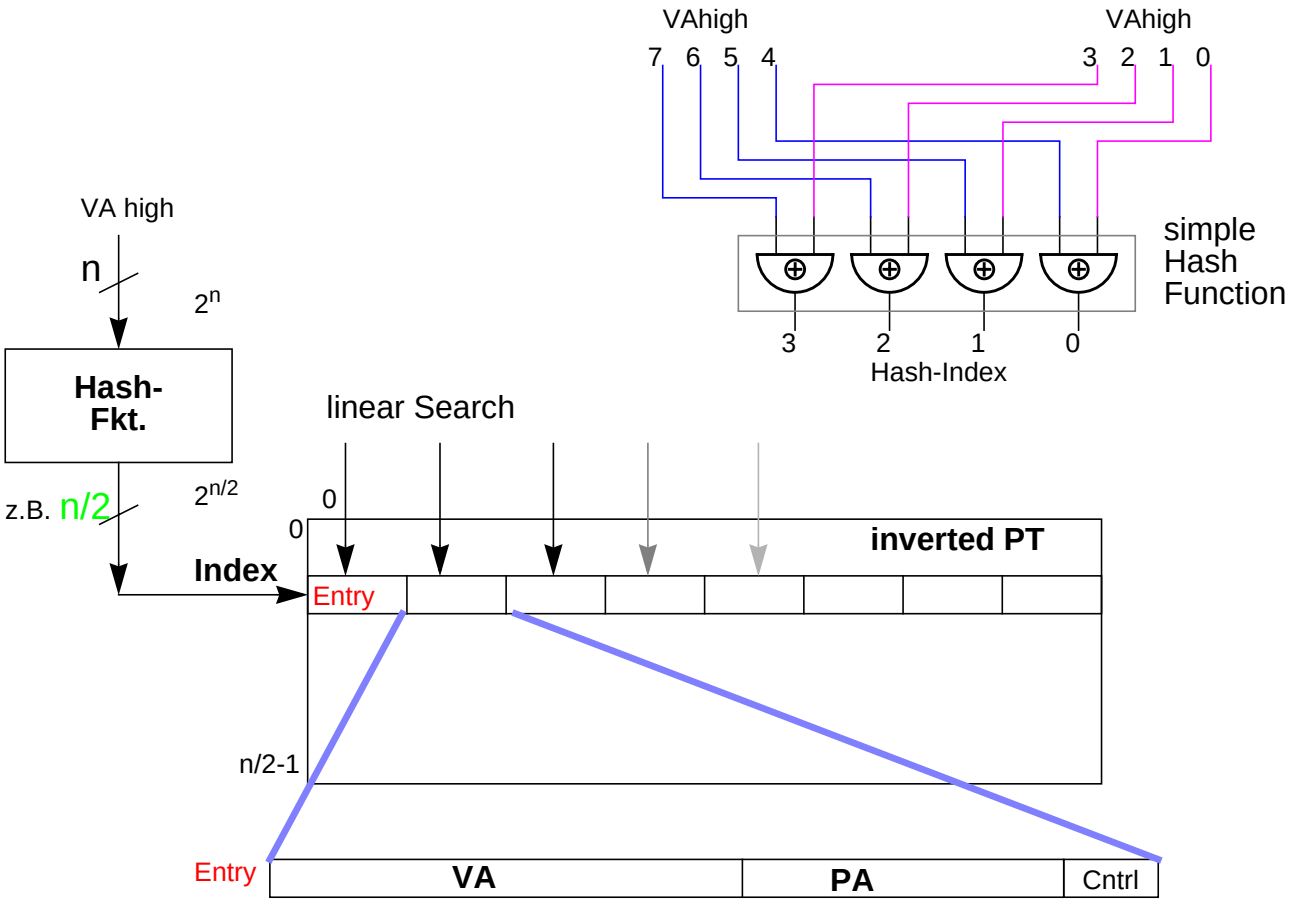
   Im Idealfall sollten sich die keys möglichst gleichmäßig über die Indices verteilen.

   - Problem: Mehrdeutigkeit der Abbildung

2. Auflösung der Kollisionen, die durch die Mehrdeutigkeit entstehen.

   a) durch anschließendes lineares Suchen

   b) durch erneutes Hashing



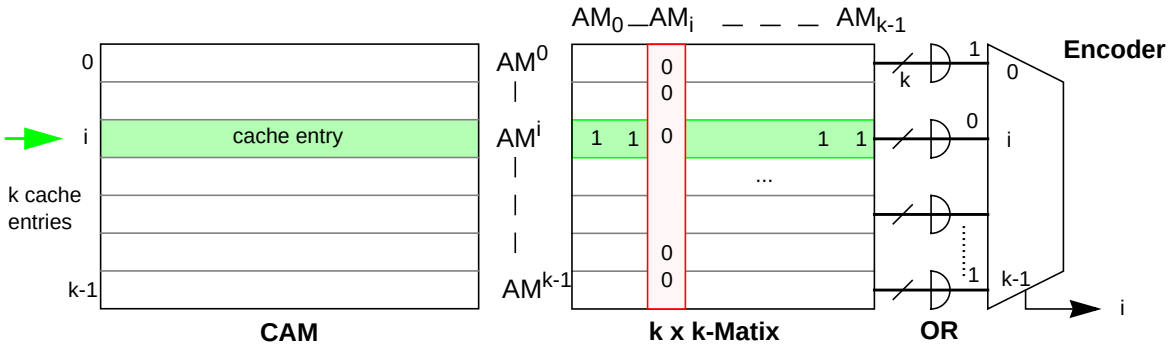
Memory Management

LRU-Verfahren

Als Beispiel für einen Algorithmus, der das LRU-Verfahren realisiert, sei ein Verfahren betrachtet, das in einigen Modellen der IBM/370-Familie angewandt wurde.

Sei CAM (**C**ontent **A**ddressable **M**emory) ein Assoziativspeicher mit k Zellen, z.B. der Cache. Zusätzlich dazu wird eine (k x k)-Matrix AM mit boolschen Speicherelementen angelegt. Jeder der 'entries' des CAM ist genau eine Zeile und eine Spalte dieser Matrix zugeordnet.

Wird nun ein entry vom CAM aufgesucht, so wird zuerst in die zugehörige Zeile der boolschen Matrix der Einsvektor e(k) und danach in die zugehörige Spalte der Nullvektor n(k) eingeschrieben (e(k) ist ein Vektor mit k Einsen; n(k) ist ein Vektor mit k Nullen). Dies wird bei jedem neuen Zugriff auf das CAM wiederholt. Sind nacheinander in beliebiger Reihenfolge alle k Zellen angesprochen worden, so enthält die Zeile, die zu der am längsten nicht angesprochenen Zelle von CAM gehört, als einzige den Nullvektor n(k).



Sei i der Index der als erste angesprochenen Zelle von CAM, und sei AM die (k x k)-Alterungsmatrix. Dann ist nach dem Ansprechen der Zelle i $AM_i = n(k)$, während alle Elemente von AM^i eins sind bis auf das Element $(AM^i)_i$, das null ist (da zunächst e(k) in die Zeile AM^i und dann in die Spalte AM_i eingeschrieben wird). Dabei bezeichnen wir die Zeilen einer Speichermatrix durch einen hochgestellten Index und die Spalten durch einen tiefgestellten Index.

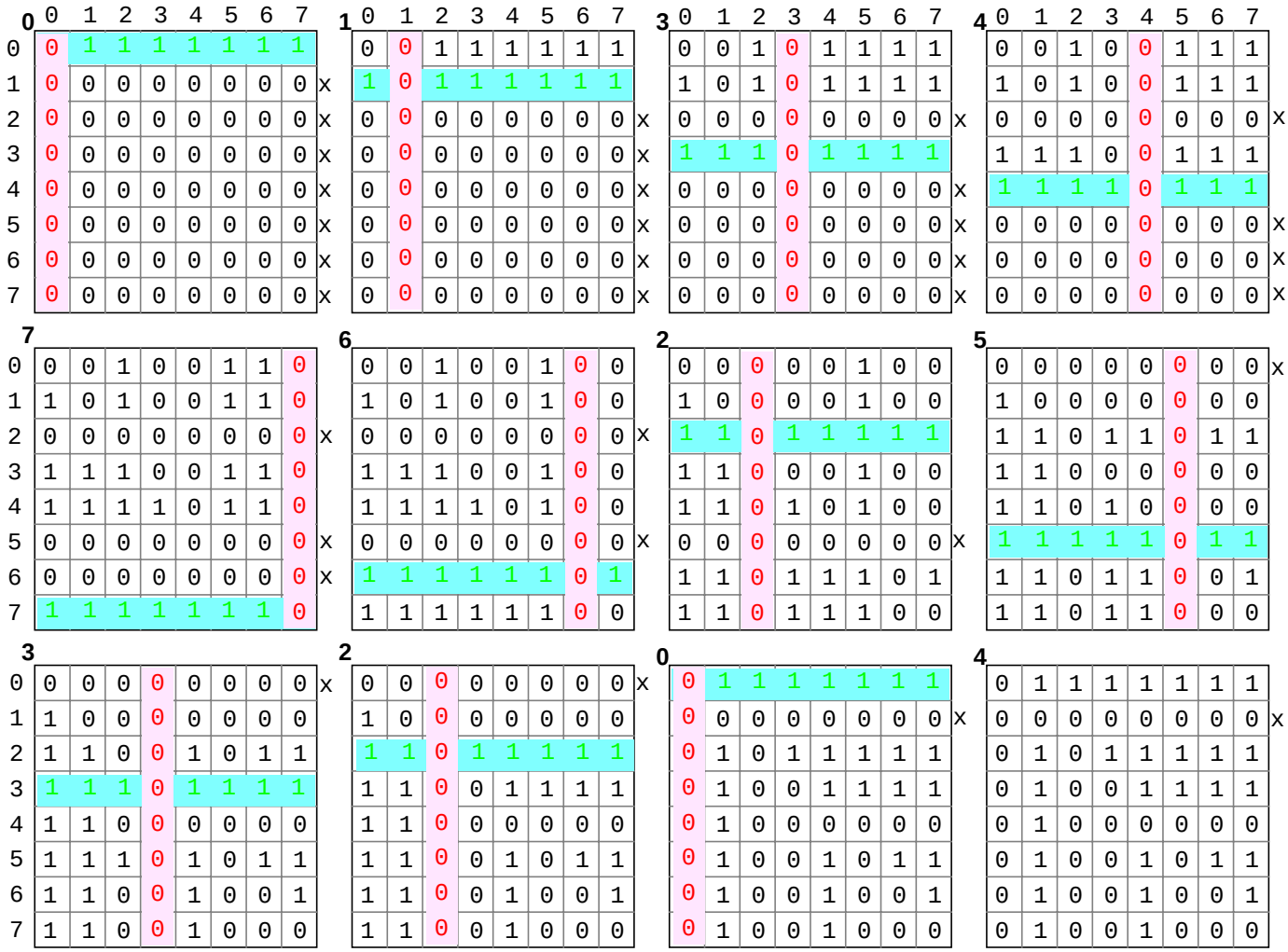
Bei jeder Referenz einer anderen Zelle von CAM wird durch das Einschreiben von e(k) in die entsprechende Zeile und nachfolgend durch das Einschreiben von n(k) in die entsprechende Spalte von AM eine der Einsen in AM^i durch Null ersetzt und eine andere Zeile mit Einsen angefüllt (bis auf das Element auf der Hauptdiagonale, das null bleibt). Damit werden nach und nach alle Elemente von AM^i durch Null ersetzt, falls die Zelle i zwischenzeitlich nicht mehr angesprochen wird. Da aber nach k Schritten nur in einer der k Zellen alle Einsen durch Nullen überschrieben sein können, müssen alle anderen Zeilen von AM noch mindestens eine Eins enthalten. Damit indiziert die Zeile von AM, die nur Nullen enthält, die LRU-Zelle (entry) des Assoziativspeichers. [Giloi, Rechnerarchitektur, Springer, 1993, pp. 130]

Memory Management

LRU-Verfahren: Beispiel

Es wird ein CAM mit 8 entries angenommen. Das kann ein vollassoziativer Cache mit 8 entries sein oder auch ein Set eines Caches mit 8 'ways' sein. Die Initialisierung der Alterungsmatrix erfolgt mit '0'. Als Beispiel für die Veränderung der Werte in AM soll die folgende Zugriffsreihenfolge betrachtet werden.

Zugriffsreihenfolge: 0, 1, 3, 4, 7, 6, 2, 5, 3, 2, 0, 4



Der Start erfolgt mit allen Einträgen als gleichalt markiert. Der erste Eintrag in Zeile und Spalte 0 läßt den Eintrag 0 altern. Es sind danach nur noch die mit x markierten Einträge zum Ersetzen zu verwenden. Sind alle Einträge einmal referenziert, so bleibt in jedem weiteren Zugriffsschritt immer nur ein Eintrag als der Älteste markiert stehen. [Tannenbaum, Modern Operating Systems, Prentice Hall, 1992, pp.111-112]

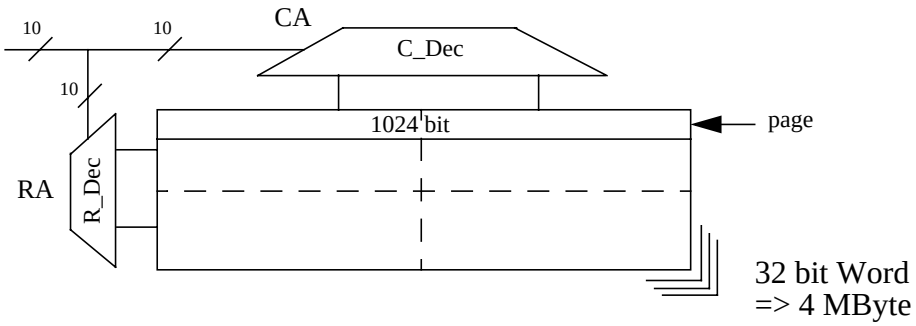
Main Memory

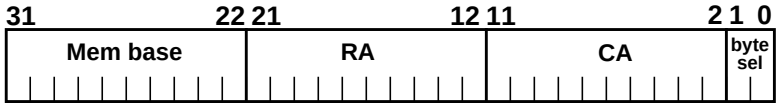
Some definitions ...

.... mode :

page mode : Betriebsart in der auf alle bits in dieser Page (RA) wesentlich schneller zugegriffen werden kann. (12-25ns t_{CAC})
Zum Zugriff wird eine neue CA und CAS benötigt.

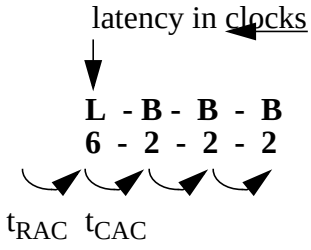
1 Mbit x 1 bit





burst mode : Holt 2^n Werte aus dem Speicher ohne eine neue CA.
CAS muß aber geschaltet werden, um den internen Zähler zu inkrementieren.

memory modell :



Main Memory

Some definitions ...

Hardware view:

asynchronous :

Without a relation to a global clock signal.
Signals propagating a net of gates behave asynchronous, because of the delay within each gate level.

synchronous:

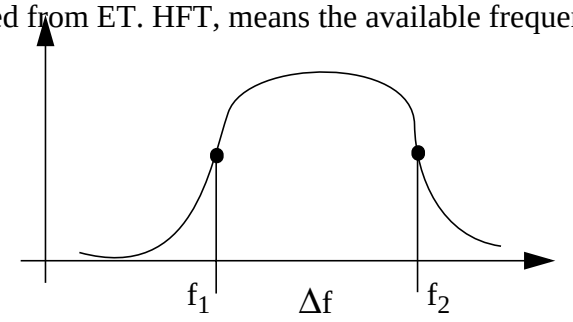
All signals and signal changes are referenced to a global clock signal.
All locations are supplied by this clock signal, it is assumed that the edge of the clock signal defines the same time at all locations

Caution : simplified view

- clock jitter
- clock skew
- clock network delay

bandwidth :

Term derived from ET. HFT, means the available frequency space Δf for a transmission channel



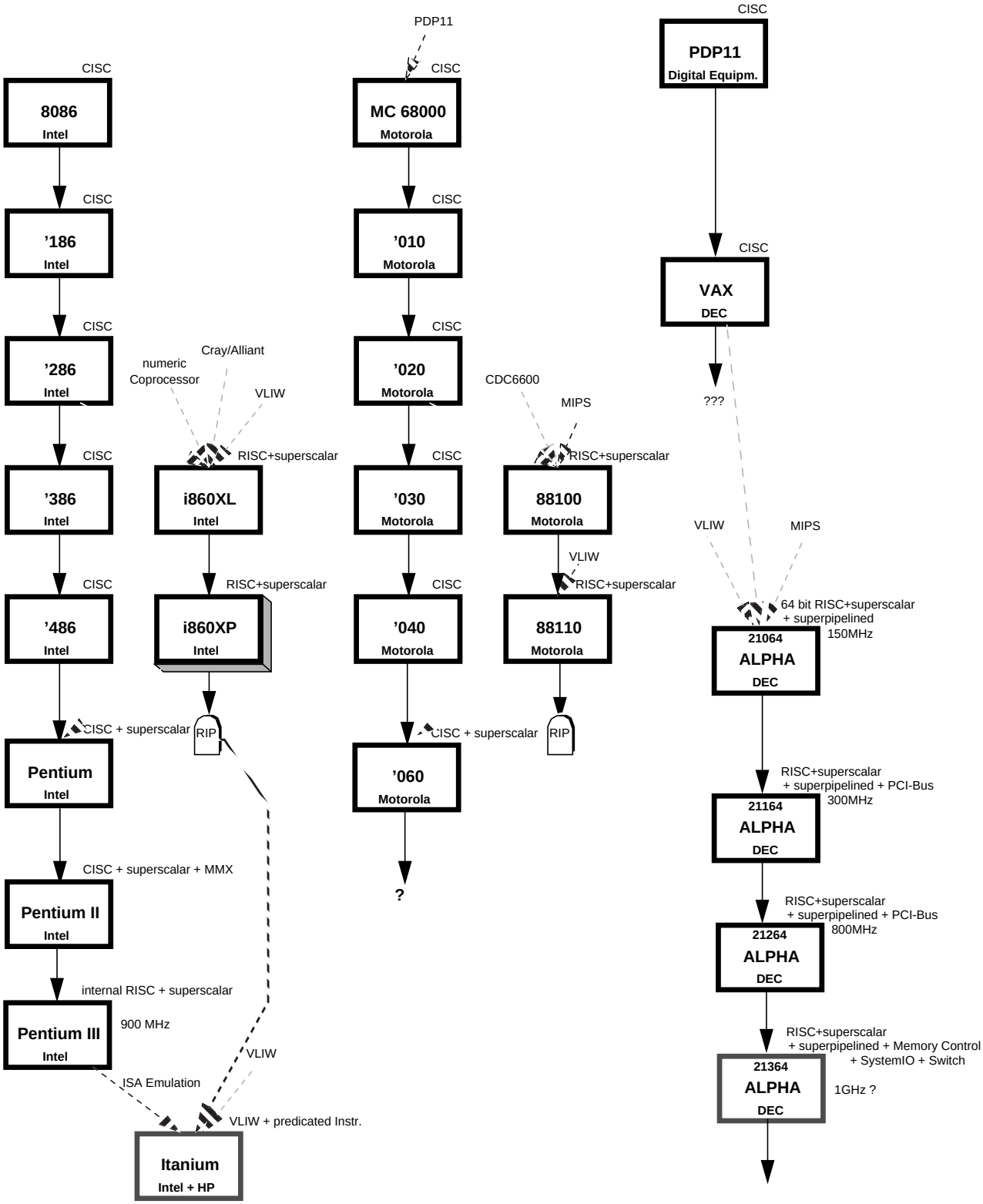
$\Delta f = f_2 - f_1$
unit [MHz]

used in TI as:
possible number of data, which can be carried through an interface, eg. bus, memory.
unit [MBytes/s]

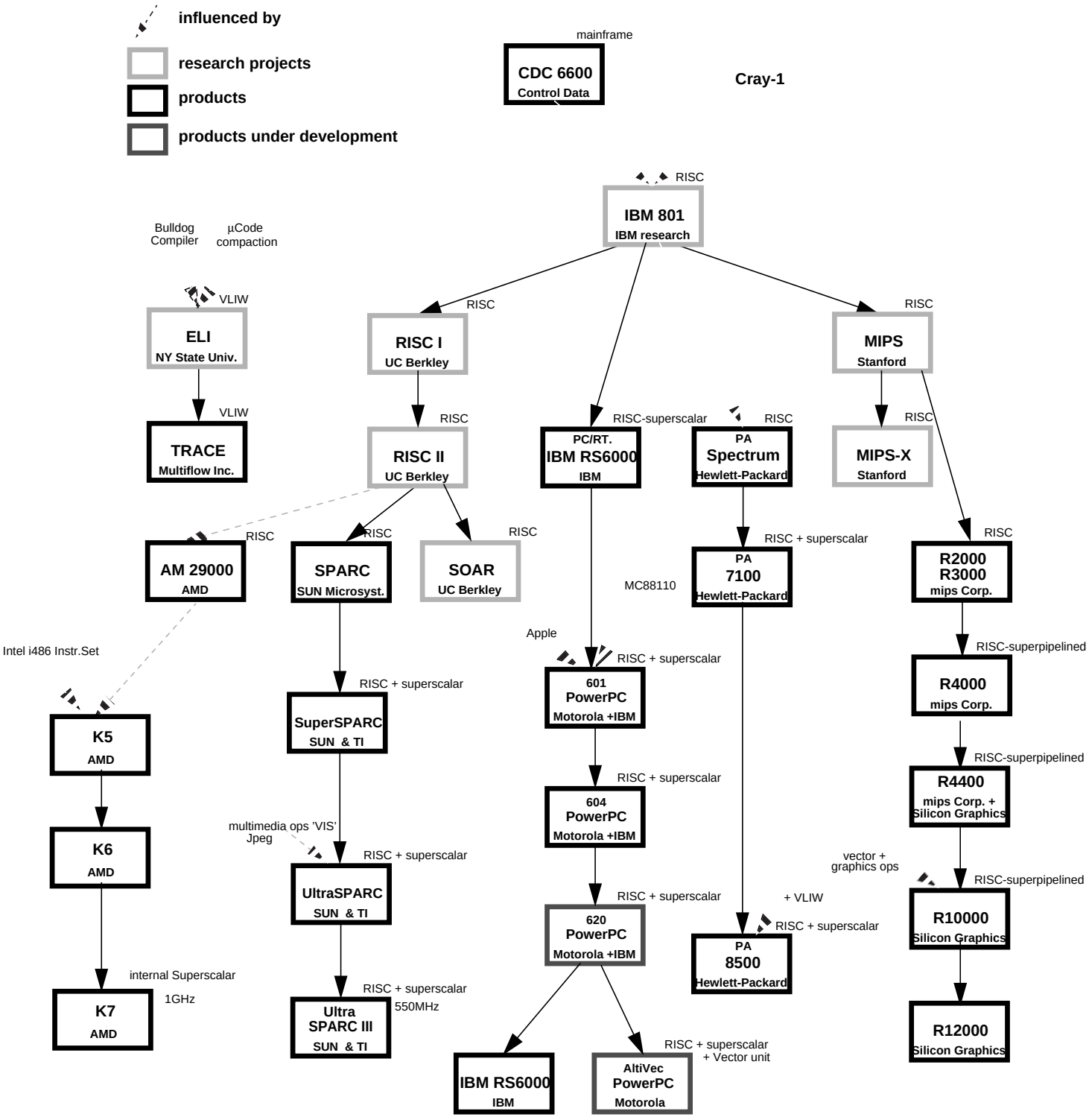
transfer rate :

Number of Data items, which are moved in one second.
unit [MBytes/s]

Development Trend- RISC versus CISC

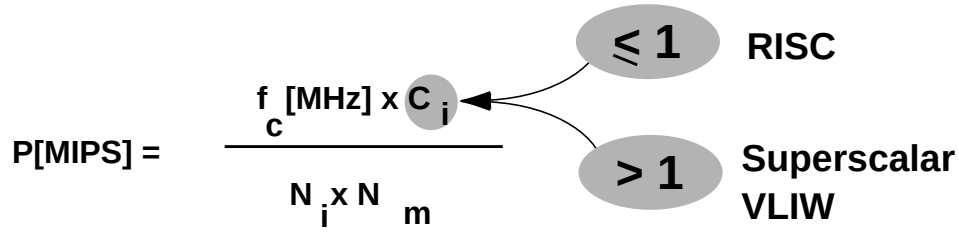


Genealogy of RISC Processors



CISC - Prozessoren

Computer Performance Equation



f_c	clock frequency
C_i	instruction count per clock cycle
N_i	average number of clock cycles per micro instruction
N_m	memory access factor

CISC - Complex Instruction Set Computer

$C_i = 1$	one Instruction per clock tick = one Operation executed by one Execution-Unit
$N_i = 5 - 7$	5-7 clock ticks required for the execution of one instruction by a number of microinstruction (Microprogram A.)
$N_m = 2 - 5$	dependent of the memory system and of the memory hierarchy

Goal : Closing the gap between High Level Language and Processor Instructions (semantic gap)

- manifold addressing modes
- microprogrammed complex instructions required a variable instruction lenght
  => varible execution time
- orthogonal instruction set : every operation matches every addressing mode

CISC - Prozessoren

CISC - Architekturen

Nachteile

- Pipelining schwierig
  - variables Instruktionsformat -> Instruction-Fetch komplex
  - unterschiedlich lange Instruction-Execution

- Memory hierachie
  - Speicherzugriff ist in Operation enthalten
  - sequentielle Instruktionsverarbeitung
  - kein prefetch

- Compilation -> Instruction-Set wird nicht genützt !

Vorteile

- Kompakter Code
  - Instruction-Cache kleiner
  - Instruction-Fetch transfer rate niedriger 1/2 Codesize of RISC (32 bit festes Format)

- Assembler -> dichter an High-Level-Language

CISC - Prozessoren

MC 68020

typical CISC - Member

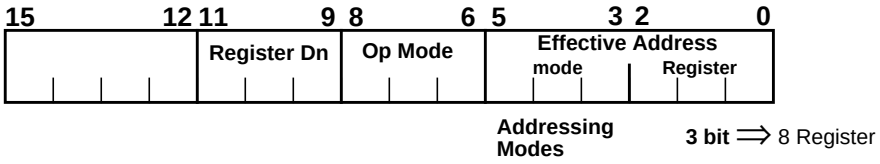
16 bit Instruction format with variable length of n times 16 bit-Units

$1 \leq n \leq 6$

8 * Data Register e.g.: Addition of Data Value ADD
8 * Address Register e.g.: Address Computation ADD**A**

Instruction-Set extendable by Coprocessor Instruction Encoding Fxxx

ADD



Operation Mode

- Byte
- Word (16 bit)
- Long (32 bit)
- Double (64 bit)

<ea> + <Dn> -> <Dn>
<Dn> + <ea> -> <ea>

ea = effective address

Effective Address Encoding Summary

- Data Register Direct
- Address Register Direct
- Address Register Indirect
- Address Register Postincrement
- Address Register Predecrement
- Address Register Displacement
- Address Register and Memory Indirect with Index
- Absolut Short (16 bit)
- Absolut Long (32 bit)
- PC-Counter Indirect with Displacement
- PC-Counter and Memory Indirect with Index
- .....

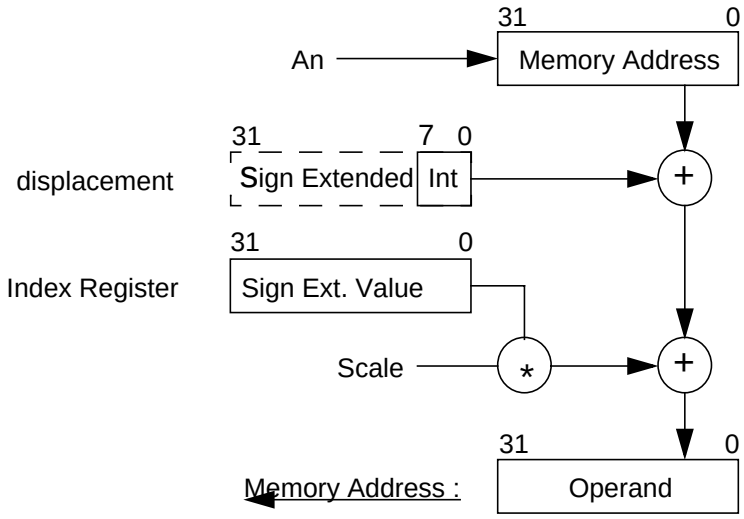
CISC - Prozessoren

MC 68020

Example of an addressing mode

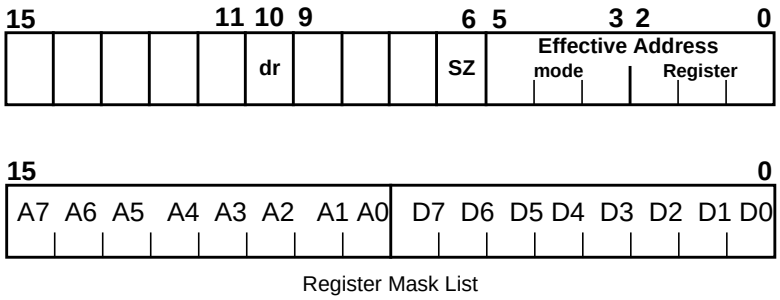
Address register indirect with index (base displacement)

$EA = (An) + (Xn) + d_8$



Example of a ’complex instruction’

MOVEM



instruction for saving the register contest

- uses postincrement addressing mode for save
- uses predecrement addressing mode for restore

CISC - Prozessoren

DBcc instructions

(-> DBcc Instruction im CPU32 User Manual)

Don't branch on condition !!!

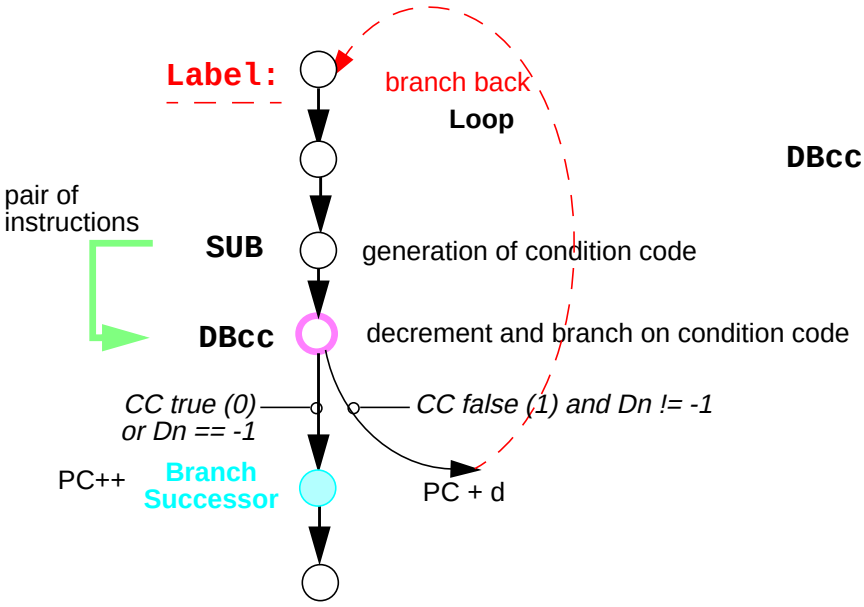
```
REPEAT
   (body of loop)
UNTIL (condition is true)
```

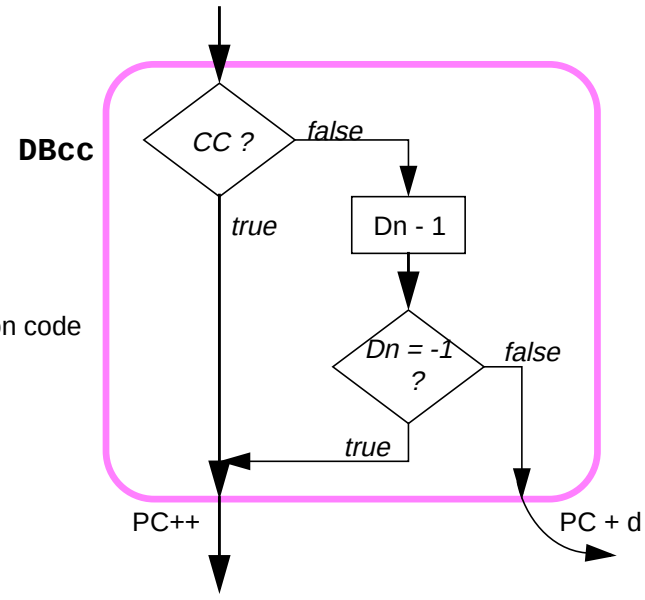
Assembler syntax:

```
DBcc Dn, <label>
```

The DBcc instruction can cause a loop to be terminated when either the specified condition CC is true or when the count held in Dn reaches -1. Each time the instruction is executed, the value in Dn is decremented by 1. This instruction is a looping primitive of three parameters: a condition, a counter (data register), and a displacemt. The instruction first tests the condition to determine if the termination condition for the loop has been met, and if so, no operation is performed. If the termination condition is not true, the low order 16 bits of the counter data register are decremented by one. If the result is - 1 the counter is exhausted and execution continues with the next instruction. If the result is not equal to - 1, execution continues at the location indicated by the current value of the PC plus the sign-extended 16-bit displacement. The value in the PC is the current instruction location plus two.

```
IF (CC == true)
    THEN PC++
    ELSE {
        Dn--
        IF (Dn == -1)
        THEN PC++
        ELSE PC <- PC + d
        }
```





Instruction format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	condition				1	1	0	0	1	register		
displacement															

Mikroprogrammierung

µ-Programmierung

Definition : ***Mikroprogrammierung ist eine Technik für den Entwurf und die Implemtierung der Ablaufsteuerung eines Rechners unter Verwendung einer Folge von Steuersignalen zur Interpretation fester und dynamisch änderbarer Datenverarbeitungsfunktionen. Diese Steuersignale, welche auf Wortbasis organisiert sind (Mikrobefehle) und in einem festen oder dynamisch änderbaren Speicher (Mikroprogrammspeicher, Control Memory oder auch Writable Control Store) gehalten werden, stellen die Zustände derjenigen Signale dar, die den Informationsfluss zwischen den ausführenden Elementen (Hardware) steuern und für getaktete Übergänge zwischen diesen Signalzuständen sorgen. [Oberschelp / Vossen]***

Das Konzept der Mikroprogrammierung wurde von M.V. Wilkes 1951 vorgeschlagen.

Man unterscheidet 2 Arten der Mikroprogrammierung:

- horizontale: die einzelnen Bits eines Mikroprogrammwortes (eine Zeile des Control Stores) entsprechen bestimmten Mikrooperationen. Diese Operationen können dann parallel angestoßen werden.
- vertikale: die Zuordnung zwischen den Bits eines Mikroprogrammwortes und den assoziierten Operationen wird durch einen sogenannten Mikrooperationscode (verschlüsselt) bestimmt.

µ-Programmierung ist der Entwurf einer Steuerungssequenz für das µ-Programmsteuerwerk.

Das µ-Programmsteuerwerk kann allgemein als endlicher Automat angesehen werden, der sich formal wie folgt beschreiben lässt:

$$A = (Q, \Sigma, \Delta, q_0, F, \delta)$$

Dessen allgem. Übergangsfunktion lautet:

$$\delta: \quad Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow Q \times (\Delta \cup \{\varepsilon\})$$

und nimmt für das Steuerwerk die spezielle Form an:

$$\delta_2: \; B^s \times (B^i \cup \{\varepsilon\}) \rightarrow B^s \times (B^o \cup \{\varepsilon\})$$

B^s state vector
B^i input vector
B^o output vector

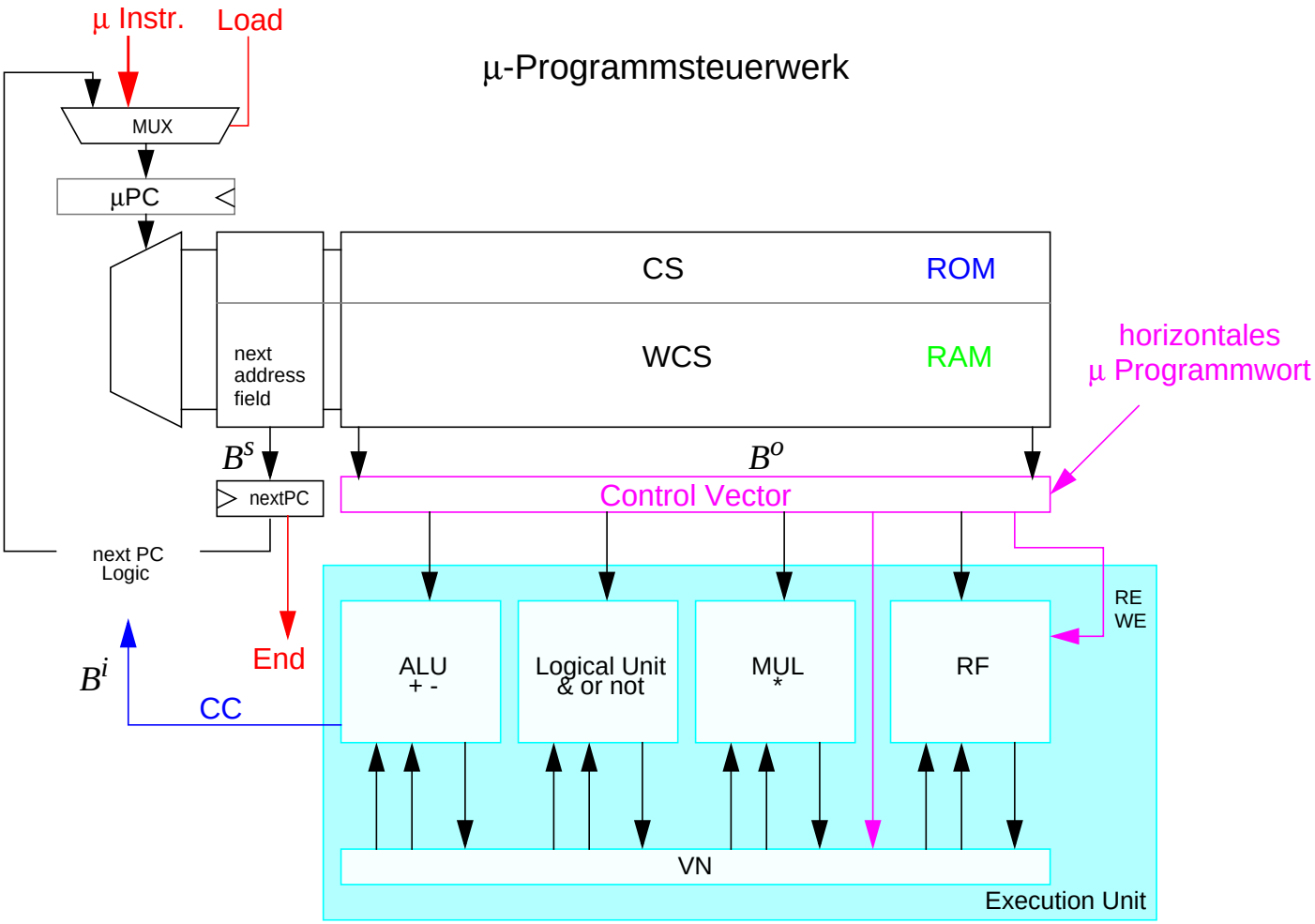
Die Berechnung des Folgezustands eines aktuellen Zustands erfolgt dabei durch die so genannte Next-State-Logik, die Berechnung des Outputs durch die Output-Logik.

Der abstrakte Begriff des endlichen Automaten ist ein entscheidendes Hilfsmittel für das Verstehen der Arbeitsweise und der Modellierung von Hardwarestrukturen. Seine Beschreibung erfolgt häufig als Graph (FSM, ’bubble diagram’), um die Zustände und die Transitionen zu veranschaulichen.

Mikroprogrammierung

µ-Programmierung

Das folgende µ-Programmsteuerwerk zeigt den prinzipiellen Aufbau. Der Control Vector stellt die Steuerungsfunktion der Execution Unit dar.



Die Mikroprogrammierung bietet folgende Vorteile:

- Implementierung eines umfangreichen Befehlssatzes (CISC) mit wenigen Mikrobefehlen bei wesentlich geringeren Kosten
- Mikroinstruktionen und damit der Maschinenbefehlssatz kann verändert werden durch Austausch des Inhaltes des Control Memory (Writable Control Store WCS)
- vereinfachte Entwicklung und Wartung eines Rechners durch deutlich geringeren Hardware-Aufwand eines Rechners

Sie hat folgenden Nachteil:

- Ausführung einer mikroprogrammierten Operation dauert länger, denn das Mikroprogramm muss schrittweise aus dem Control Memory gelesen werden. Für jeden Befehl sind meist mehrere ROM-Zugriffe erforderlich.

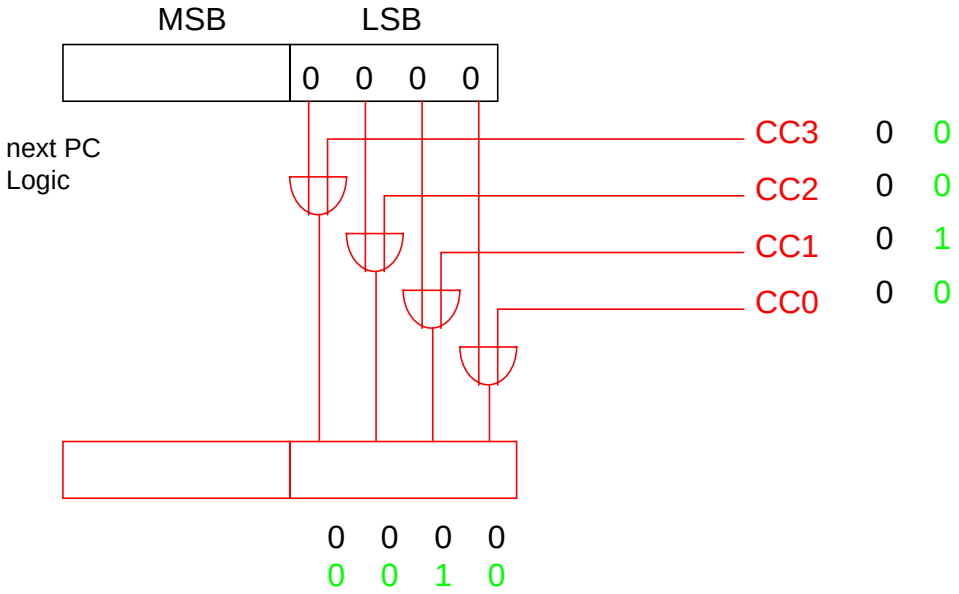
Mikroprogrammierung

µ-Programmierung

Wenn das Steuerwerk schnell (im darauffolgenden Takt) auf die CC-Signale reagieren soll, wird die Next-PC Logik nicht wie sonst üblich durch einen Addierer ausgeführt, sondern durch eine einfache Oder-Logik, die die LSBs der Folgeadresse modifizieren. Verwendet man als Folgeadresse eine Adresse mit den vier LSBs auf "0", so können die Condition Codes 0..3 diese bits zu "1" werden lassen, womit man einen 16-way branch realisieren kann.

Dadurch sind die Adressen der µ-Instruktionen nicht mehr linear im Speicher angeordnet sondern können beliebig verteilt werden, da jede µ-Instruktionen ihre Folgeadresse im WCS mitführt.

Benutzt man eine Folgeadresse mit z.B. 0010 als LSBs, so kann man damit den CC1 ausmaskieren, da er durch die Oder-Funktion nicht mehr wirksam werden kann.



Die µ-Programmierung solcher Steuerwerke auf der bit-Ebene ist natürlich viel zu komplex, als dass man sie von Hand durchführen könnte. Verwendet wird üblicherweise eine symbolische Notation, ähnlich einem Assembler. Diese Notation wird dann durch einen Mikro-Assembler in die Control Words des WCS umgesetzt.

Die explizite Sichtbarkeit des Control Wortes der Execution-Hardware bildet die Grundlage der VLIW-Architekturen.

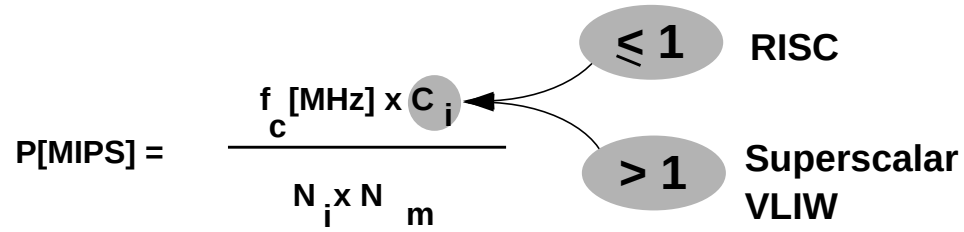
Der Test und die Verifikation der µ-Programme ist sehr aufwendig, da sie direkte Steuerungsfunktionen auf der HW auslösen, deren Beobachtung meist nur durch den Anschluss von LSAs (Logic State Analyzer) möglich ist.

Die µ-Programmierung wurde wegen der höheren Geschwindigkeit bei der Einführung der RISC-Rechner durch festverdrahtete Schaltwerke (PLAs, Programmable Logic Arrays) abgelöst.

weitere Informationen in: W.Oberschelp, G.Vossen, Rechneraufbau und Rechnerstrukturen, 7.Aufl., R.Oldenbourg Verlag, 1998

RISC

Computer Performance Equation



f_c	clock frequency
C_i	instruction count per clock cycle
N_i	average number of clock cycles per instruction
N_m	memory access factor

The RISC design philosophy can be summarized as follows:

- pipelining of instructions to achieve an effective single cycle execution (small N_i)
- simple fixed format instructions (typically 32 bits) with only a few addressing modes
- hardwired control of instruction decoding and execution to decrease the cycle time
- load-store architectures keep the memory access factor N_m small
- Migration of functions to software (Compiler)

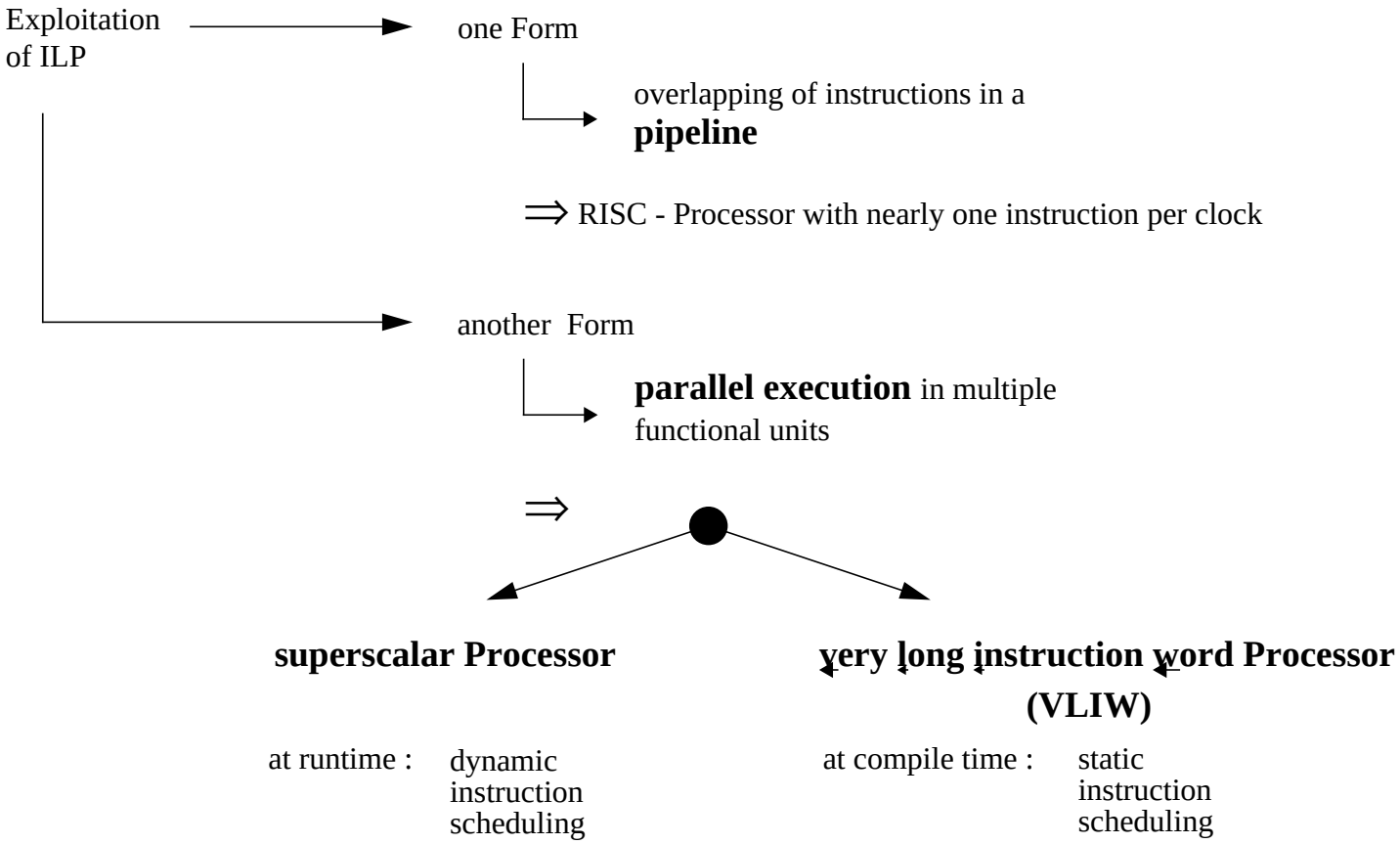
The goal of the reduced instruction set is to maximize the speed of the processor by getting the software to perform infrequent functions and by including only those functions that yield a net performance gain. Such an instruction set provides the building blocks from which high-level functions can be synthesized by the compiler without the overhead of general but complex instructions [fujitsu]

The two basic principles that are used to increase the performance of a RISC processor are:

- **pipelining and**
- **the optimization of the memory hierarchy.**

ILP Instruction level parallelism

Definition : ***It is the parallelism among instructions from 'small' code areas which are <u>independent</u> of one another.***



ILP Instruction level parallelism

Availability of ILP

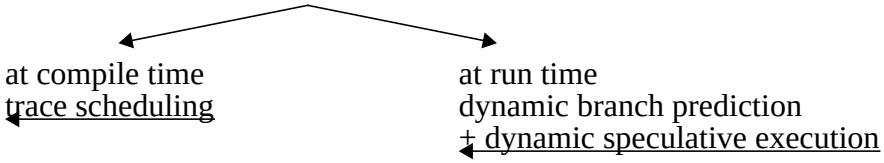
’small’ code area

1. **basic block** : a straight-line code sequence with no branches in except to the entry and no branches out except the exit
   ILP small ! ~ 3,4; <6

2. **multiple basic blocks**

   a) **loop iterations**
   Loop level Parallelism
   stems from ’Structured Data Type’ Parallelism
   Vector Processing

   b) **speculative execution**



Techniques to improve C_i (instruction count per clock cycle)

- Loop unrolling
- pipeline scheduling
- dynamic instruction scheduling; scoreboarding
- register renaming
- dynamic branch prediction
- multiple instruction issue
- dependence analysis; compiler
- instruction reordering
- software pipelining
- trace scheduling
- speculative execution
- memory disambiguation; dynamic - static

ILP

Example of Instruction Scheduling and Loop Unrolling

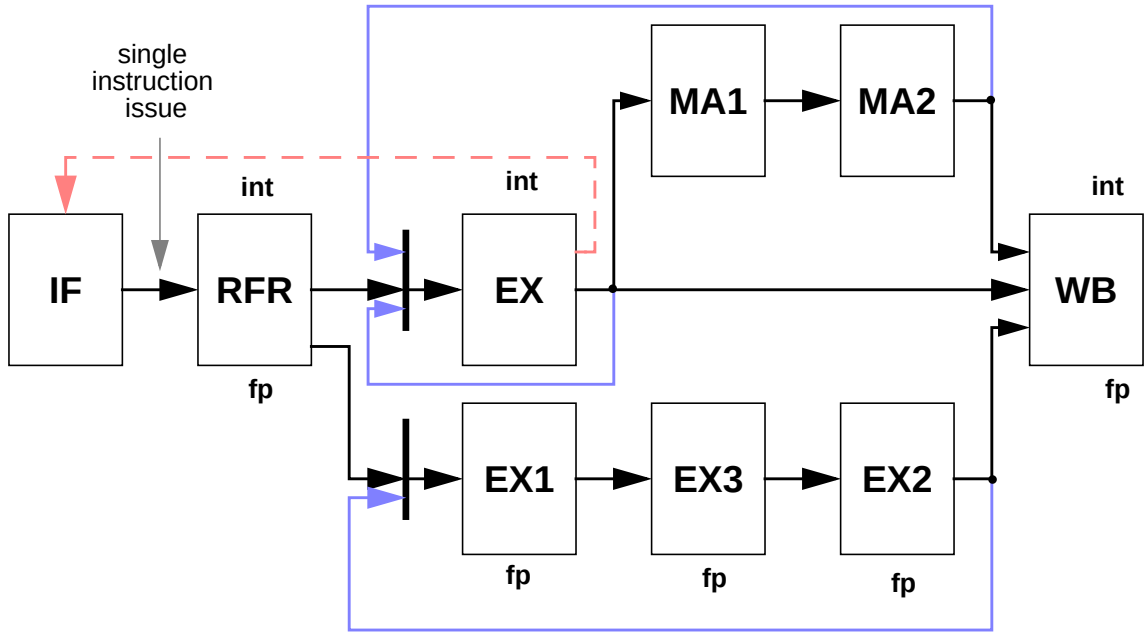
Calculate the sum of the elements of a vector with length 100

$$sum = \sum_{i=0}^{99} a_i$$

C-Program

```
int main() {
double a[100];
double sum = 0.0;
int i;

   for (i=0;i<100;i++)
      sum += a[i];
   f(sum);                 // using sum in f avoids heavy optimization
}                          // which results in "do nothing"
```

Suppose a basic 4-staged pipeline similar to Exercise 2. The pipeline is interlocked by hardware to avoid flow hazards. Forwarding pathes for data and branch condition are included. The two stages for cache access are only active for LD/ST instructions. The Cache is non-blocking. Misses queue up at the bus interface unit which is not shown in the block diagram. The cache holds four doubles in one cache entry. The cache entry is filled from main memory using a burst cycle with 5:1:1:1. Integer instructions execute in one clock, fp operations ADD and MUL requires 3 clocks. The bus-clock is half of the processor clock.



Block-Diagram of a simple pipelined RISC

ILP

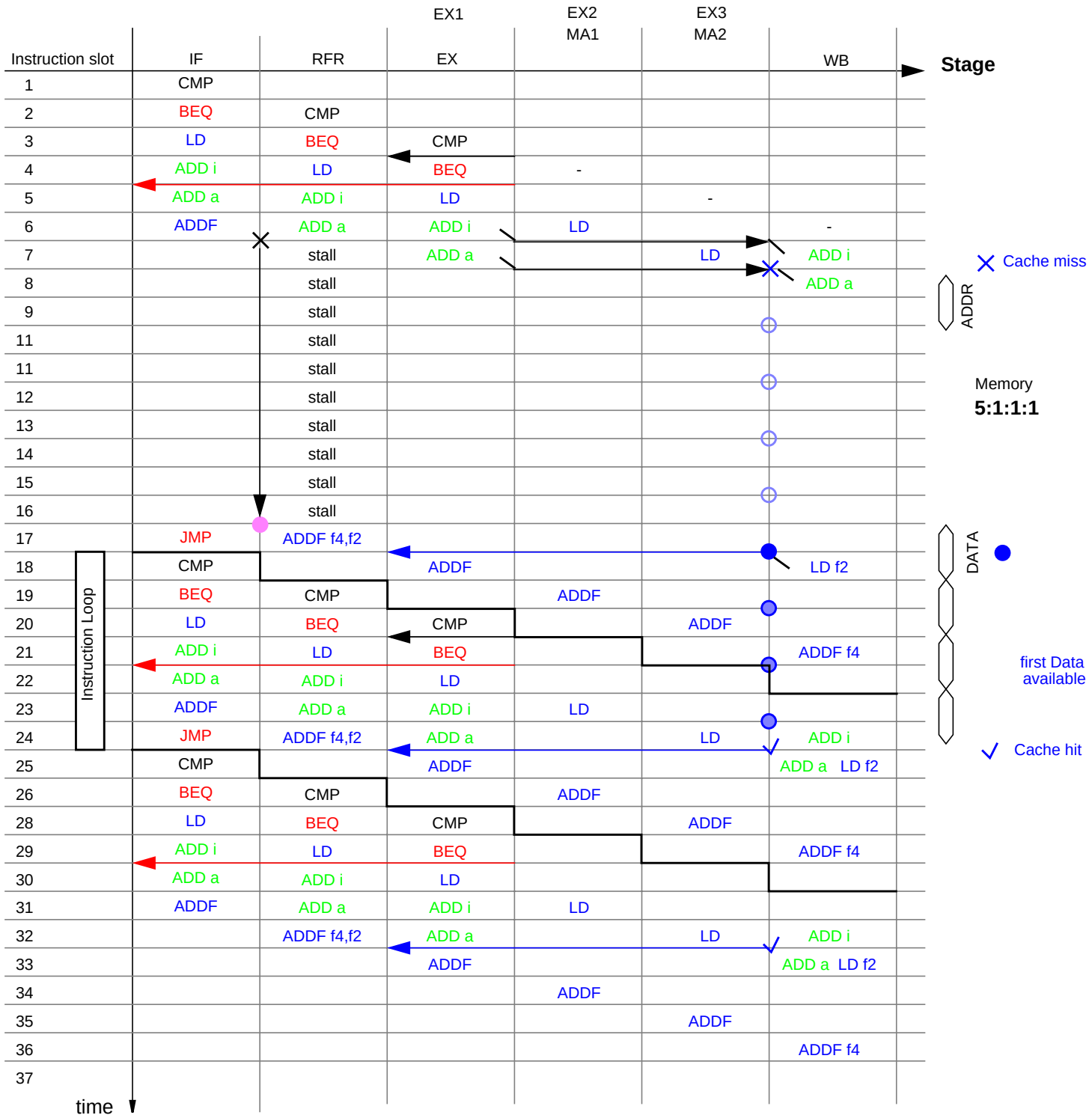
Example of Instruction Scheduling - Assembler Code

```
// Initialization

R0 = 0;             // always zero
R5 = 99;            // endcount 100-1
R4 = 0;             // loop index
R3 = 1000;          // base address of A
R2 = 800;           // address of sum
                    // fkt parameters are normally stored on stack !!

F4 = 0.0;;          // Accumulator for sum
F2 = 0.0;           // Register for loading of Ai

// Assembler Code

Lloop:  CMP R4, R5 -> R6;          // generate CC and store in R6
        BEQ R6, Lend;              // branch pred. "do not branch"
        LD  (R3) -> F2;            // load Ai
        ADDF F2,F4 -> F4;          // accumulation of sum
        ADDI R4, #1 -> R4;         // loop index increment
        ADDI R3, #8 -> R3;         // addr computation for next element
        JMP Lloop;
Lend:   ST F4 -> (R2);
```

It should be mentioned here that local variables of procedures are normally stored in the activation frame on the stack. The addressing of local variables can be found in the assembler code of the compiled C-routines. For this example, a simplified indirect memory addressing is used.

ILP

Example of Instruction Scheduling without Loop Unrolling



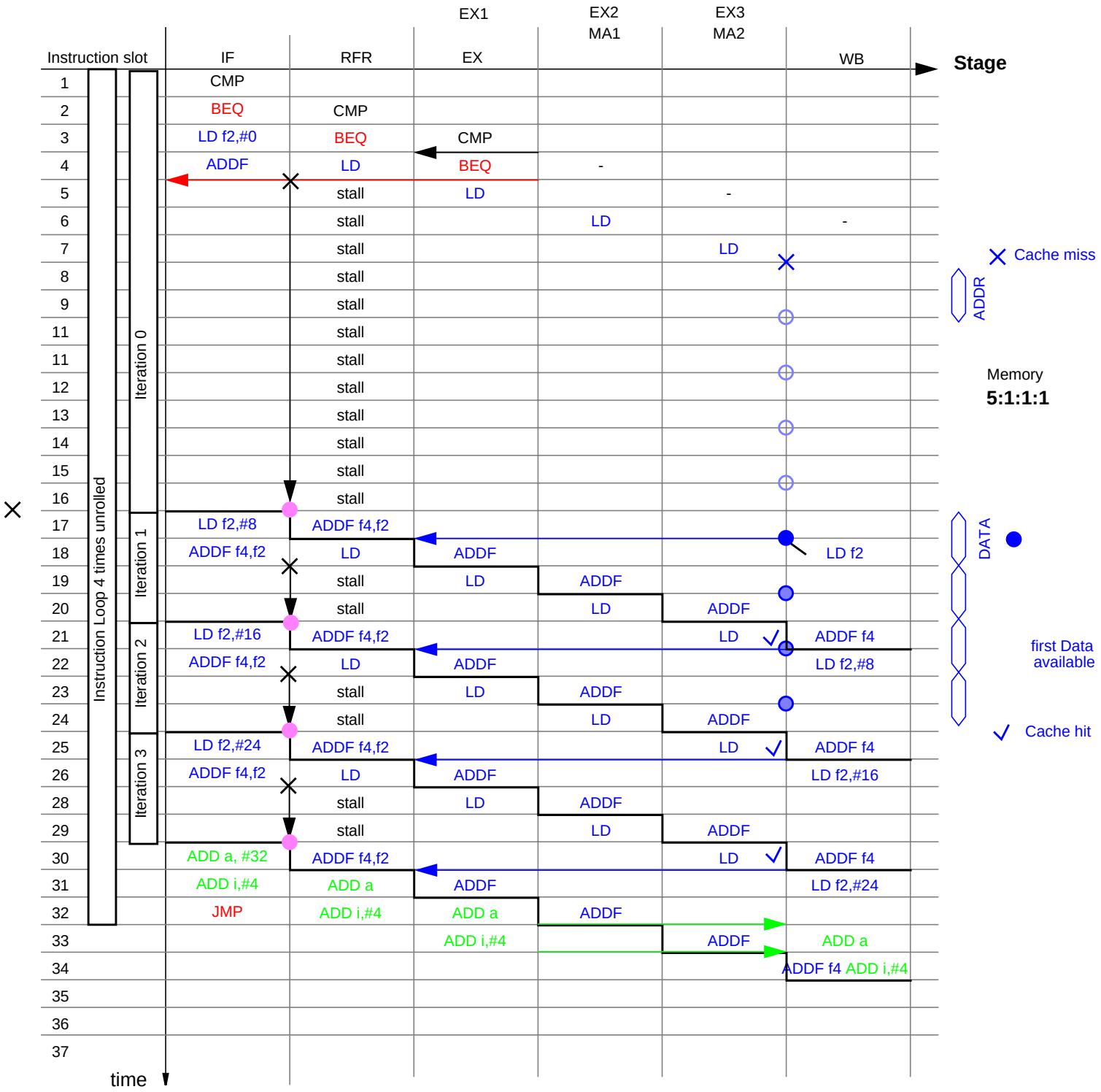
The successor of a Bcc instruction is statically predicted and speculatively executed and can be annulled if a wrong prediction path is taken.

The execution time of 100 element vector accumulation can be calculated to

100 / 4 x (17 + 3 x 7) = 950 CPU clocks

ILP

Example of Instruction Scheduling with Loop Unrolling



Load instructions are not scheduled and interations do not overlapp.

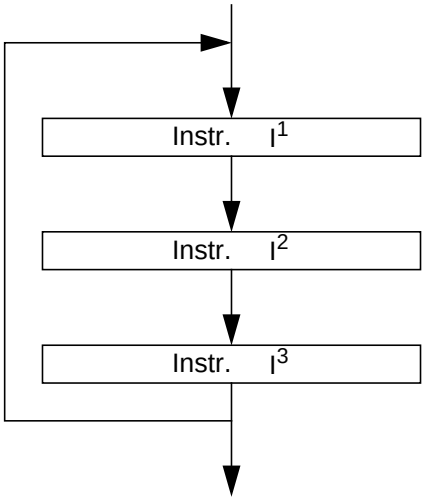
The execution time of 100 element vector accumulation can be calculated to:

100 / 4 x 32 = 800 CPU clocks

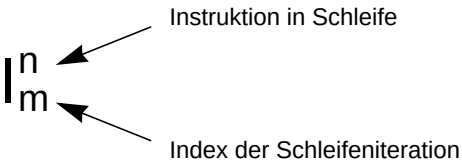
ILP

Software - Pipelining

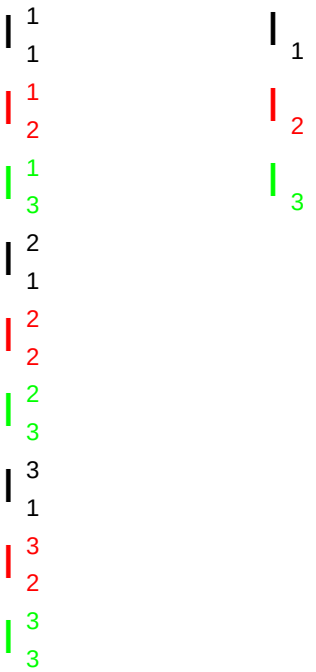
Software-Pipelining überlappt die Ausführung von aufeinanderfolgenden Iterationen in ähnlicher Form wie eine Hardware-Pipeline.



Durch das Software-Pipelining wird die Parallelität zwischen unabhängigen Schleifeniterationen ausgenützt.

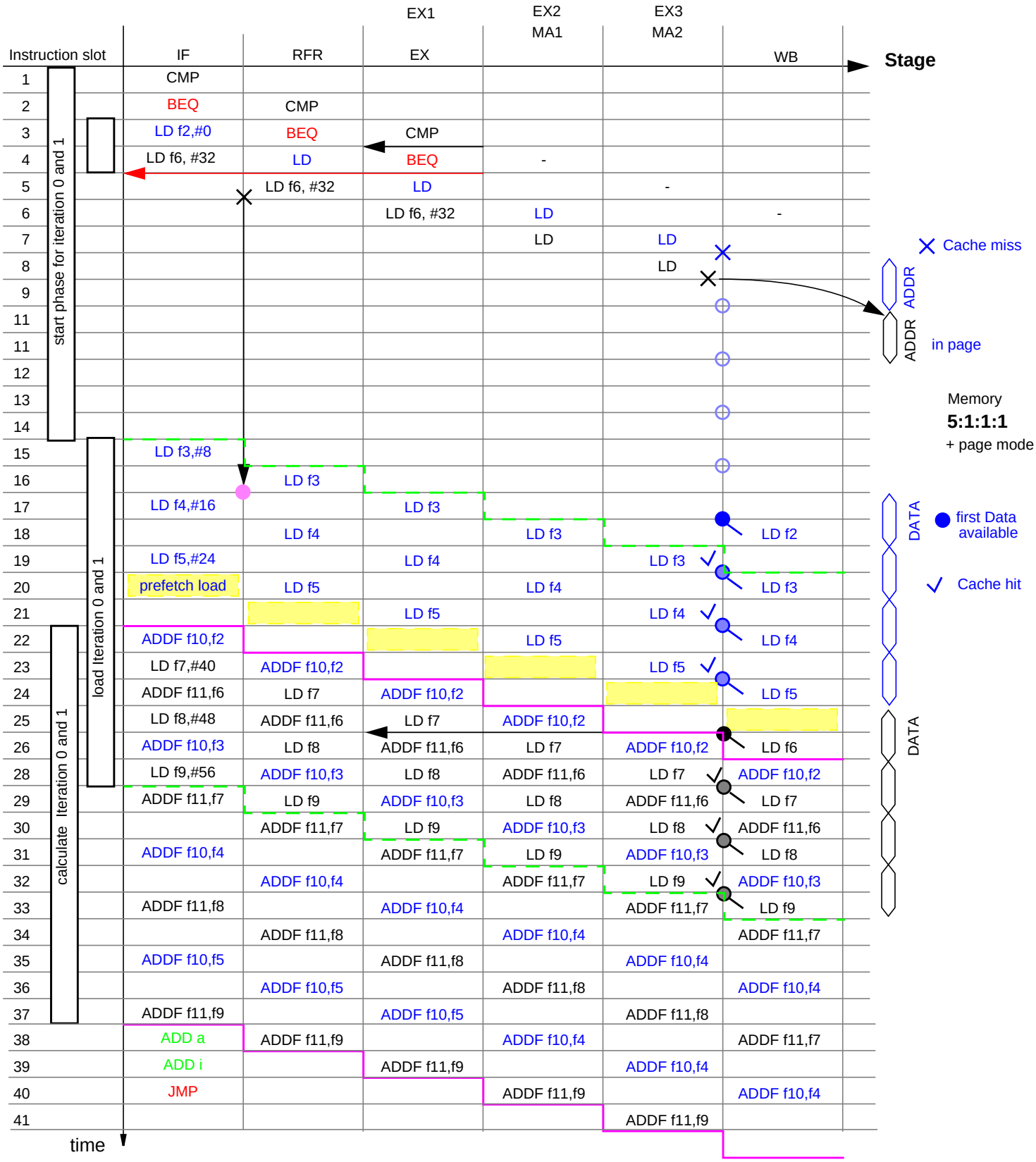


Anordnung der Instruktionen.



ILP

Example of Instruction Scheduling with Loop Unrolling and Software Pipelining



The loop is four times unrolled and instructions from different iterations are scheduled to avoid memory and cache accesss latencies. Scheduling of load instructions requires a non-blocking cache and a BUI which can handle multiple outstanding memory requests. The latency of the first start-up phase can only be avoided by scheduling the load into code before the loop start (difficult). Execution time: ~ (100/4 / 2 * 20) +(14 + 7) + 6 = 277

ILP

Assembler without optimization compiled by gcc -S <fn.c>

```
        .file   "unroll.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC0:
        .long   0x0
        .long   0x0
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save %sp,-928,%sp
        !#PROLOGUE# 1
        sethi %hi(.LLC0),%o0
        ldd [%o0+%lo(.LLC0)],%o2
        std %o2,[%fp-824]
        st %g0,[%fp-828]
.LL2:
        ld [%fp-828],%o0
        cmp %o0,99
        ble .LL5
        nop
        b .LL3
        nop
.LL5:
        ld [%fp-828],%o0
        mov %o0,%o1
        sll %o1,3,%o0
        add %fp,-16,%o1
        add %o0,%o1,%o0
        ldd [%fp-824],%f2
        ldd [%o0-800],%f4
        faddd %f2,%f4,%f2
        std %f2,[%fp-824]
.LL4:
        ld [%fp-828],%o1
        add %o1,1,%o0
        mov %o0,%o1
        st %o1,[%fp-828]
        b .LL2
        nop
.LL3:
.LL1:
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
        .ident  "GCC: (GNU) 2.7.2.2"
```

ILP

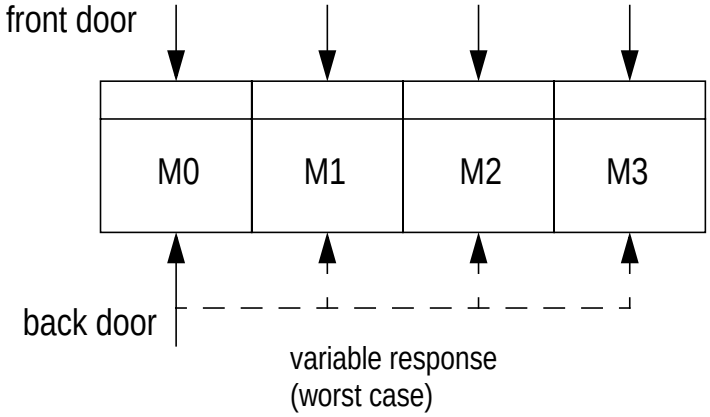
Assembler with optimization compiled by gcc -S -o3 -loop_unroll <fn.c>

```
        .file   "unroll.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC1:
        .long   0x0
        .long   0x0
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save %sp,-912,%sp
        !#PROLOGUE# 1
        sethi %hi(.LLC1),%o2
        ldd [%o2+%lo(.LLC1)],%f4
        mov 0,%o1
        add %fp,-16,%o0
.LL11:
        ldd [%o0-800],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-792],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-784],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-776],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-768],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-760],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-752],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-744],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-736],%f2
        faddd %f4,%f2,%f4
        ldd [%o0-728],%f2
        faddd %f4,%f2,%f4
        add %o1,10,%o1
        cmp %o1,99
        ble .LL11
        add %o0,80,%o0
        std %f4,[%fp-16]
        call f,0
        ldd [%fp-16],%o0
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
        .ident  "GCC: (GNU) 2.7.2.2"
```

VLIW Concepts

- long (multiple) instruction issue
- static instruction scheduling (at compile time)
  - highly optimizing compiler (use of multiple basic blocks)
- simple instruction format

  => simple Hardware structure
  - synchronous operation (global clock)
    - pipelined Funktion-Units (fixed pipeline length !)
  - resolve of resource hazards and data hazards at compile time
- control flow dependencies => limit of performance
- unpredictable latency operation
  - memory references ----> fixed response
  - dma, interrut (external)
- numerical processors



Advantages

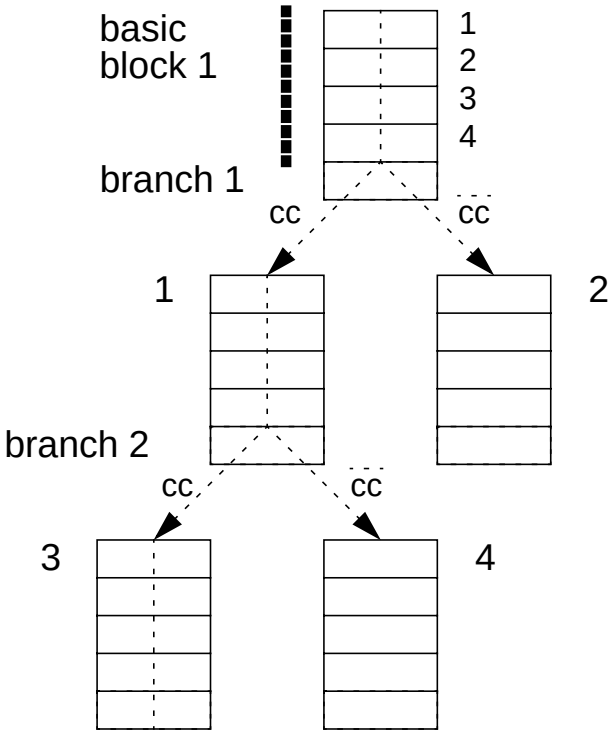
- max n times performance using n Funktional -Units
- simple Hardware-Architecture

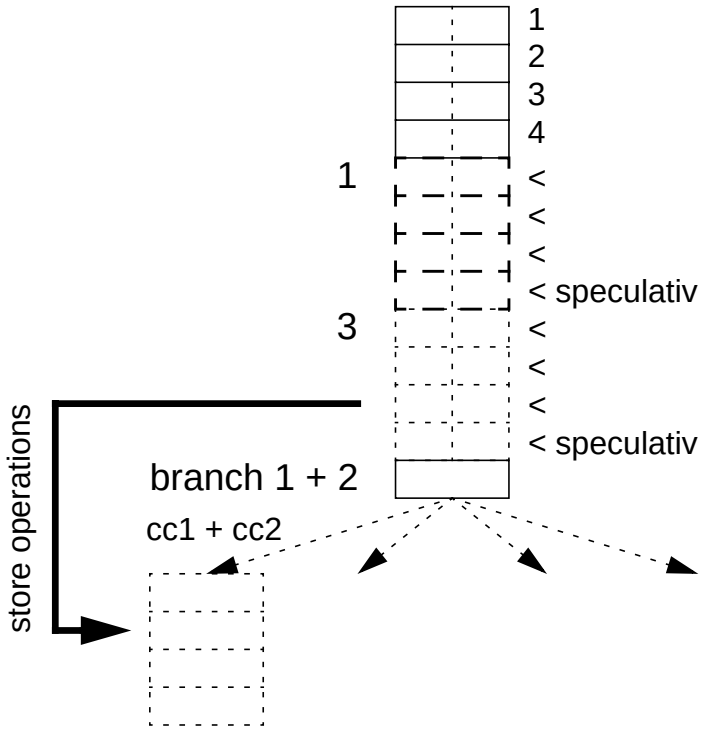
Disadvantages

- Mix of Functional-Units is Application dependent
- multiple Read/Write Ports
- code explosion
- static order
- stop of latency operations

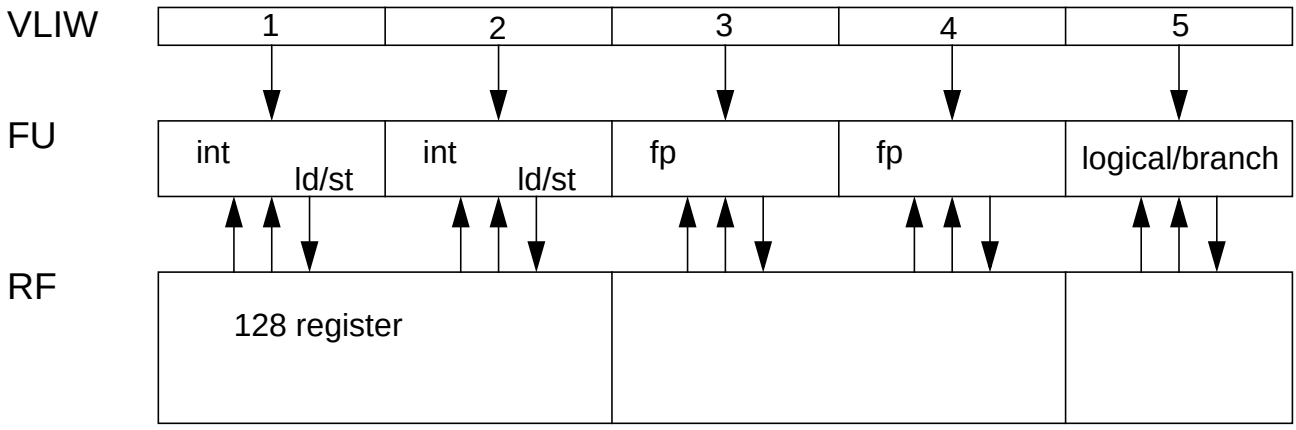
VLIW Concepts

VLIW : Trace Scheduling
Speculative Execution





resource independant-
data indipendant instruction



Superscalar Processors

- multiple instruction issue

- dynamic instruction scheduling (at run time !)
  - hardware resource to control dynamic instruction scheduling

  -
  - scoreboard (CDC 6600) ①
    [Th64] Thornton, James.E., Parallel Operation in the Control Data 6600, Proc. AFIPS, 1964

  - Tomasulu algorithm (IBM 360/91)
    [Tomasulu, R.M., An Efficient Alogrithm for Exploiting Multiple Arithmetic Units, IBM Journal, Vol. 11, 1967]

① centralized Hardware-Structure allowing

’in order’ issue

’out of order’ execution *

’out of order’ completion

* no structural Hazard --> enough Functional-Units
no data dependency

\+  WAR Hazard Anti - Dependency

\+  WAW Hazard Output - Dependency

With the multiplicity of functional units, and of operand registers and with a simple and highly efficient addressing scheme, a generalized queue and reservation scheme is practical. This is called a **scoreboard**.

The scoreboard maintains a running file of each central register, of each functional unit, and of each of the three operand trunks (source and destination busses) to and from each unit. [Th64]

Hazard detection and resolution

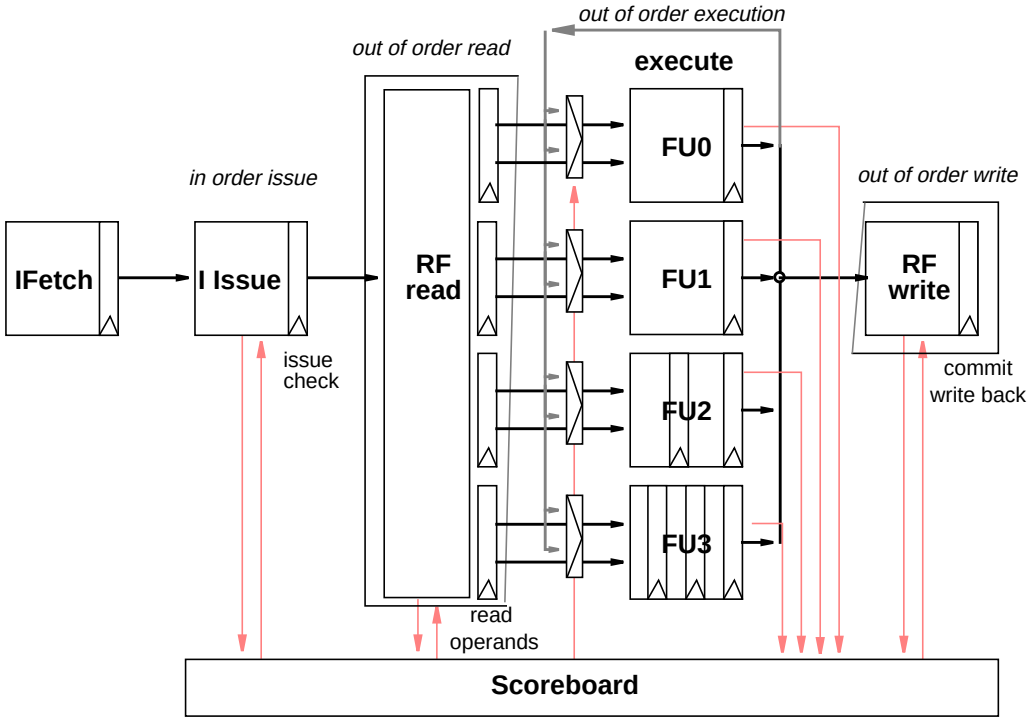
construction of data dependencies
check of data dependencies
check of resource dependencies

Hazard avoidance by stalling the later Instructions

Superscalar Processors

Dynamic Scheduling with a Scoreboard

Pipelined Processor with multiple FUs



1. I Issue: If a functional unit for the instruction is free and no other active instruction has the same destination register, the scoreboard issues the instruction to the FU and updates its internal data structure.By ensuring that no other active FU wants to write its result into the destination register WAW-Hazards are avoided. The instruction isssue is stalled in the case of a WAW-Hazard or a busy FU.
2. Read operands: The scoreboard monitors the availability of the source operands. A source operand is available, if no earlier issued **active** instruction is going to write this register. If the operands are available, the instruction can proceed. This scheme resolves all RAW-Hazards dynamically. It allows instructions to execute ’out of order‘.
3. Execution: The required FU starts the execution of the instruction. When the result is ready, it notifies the scoreboard that it has completed execution. If an other instruction is waiting on this result, it can be forwarded to the stalled FU.
4. Commit: On the existance of a WAR-Hazard, the write back of the instruction is stalled, until the source operand is read by the dependend instruction (preceeding instruction in the order of issue).
[Hennessy, Patterson, Computer Architecture: A Quantitative Approach, 2. Ed. 1996]

Stage	Wait until	Checks	Bookkeeping	Operations
I Issue activate instruction	FU available and no more than one result assigned to Rdest	FU ? available Fd ? not busy	(FU) <-- busy (Fd) <-- busy	Fd(FU) <-- ’D’ Fs1(FU) <-- ’S1’ Fs2(FU) <-- ’S2’ Op(FU) <-- ’opc’
read operands	Source operands available	Fs1(FU) ? valid Fs2(FU) ? valid	(Fs1) <-- read (Fs2) <-- read	s1(FU) <-- (Fs1) s2(FU) <-- (Fs2)
execute				R <-- s1(FU) op s2(FU)
write back deactivate instruction	Result ready and no WAR-Hazard	Fd ? read	(FU) <-- free (Fd) <-- valid	Fd <-- R

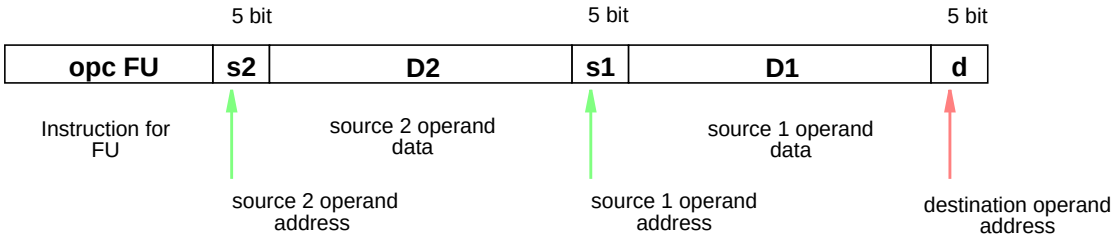
Superscalar Processors

Dynamic Scheduling with Tomasulu (out of order execution)

The main idea behind the Tomasulu algorithm is the introduction of reservation stations. They are filled with instructions from the decode/issue unit without check for source operand availability. This check is performed in the reservation station logic itself. This allows issueing instructions whose operands are not readily computed. Issuing this instruction (which would normally block the issue stage) to a reservation station shuffles the instruction besides and give the issue unit the possibility to issue the following instruction to another free reservation station.

A reservation station is a register and a control logic in front of a FU. The register contains the register file source operand addresses, the register file destination address, the instruction for this FU and the source operands. If the source operands were not avaiable, the instruction waits in the reservation station for its source operands. The control logic compares the destination tags of all result busses from all FUs. If the destination tag matches to a missing operand address, this data value is taken from the result bus into the reservation stations operand data field. A reservation station can forward its instruction to the execution FU, when all its operands are available (data flow synchronization).

A structure view of the data pathes can be found in the powerPC 620 part.



Superscalar Processors

FU Synchronization

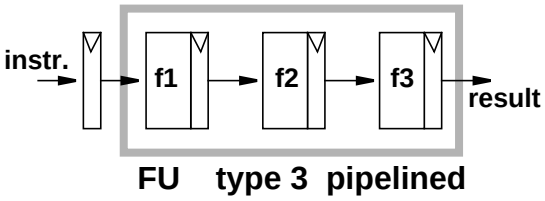
Definition

A **functional unit** is a processing element (PE) that computes some output value based on its input [Ell86]. It is a part of the CPU that must carry out the instructions (operations) assigned to it by the instruction unit. Examples of FUs are adders, multipliers, ALUs, register files, memory units, load/store units, and also very complex units like the communication unit, the vector unit or the instruction unit. FUs with internal state information are carriers of data objects.

In general, we can distinguish five different types of FUs:

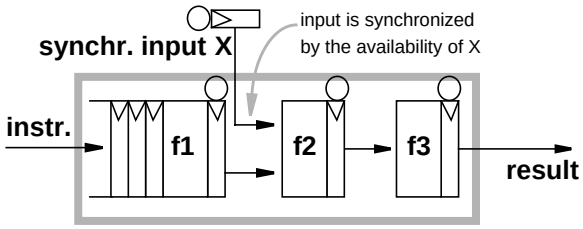
1. FU with a single clock tick of execution time
2. FU with n clock ticks of execution time, nonpipelined
3. FU with n clock ticks of execution time, pipelined
4. FU with variable execution time, nonoverlapped
5. FU with variable execution time, overlapped

The example of two FUs given in the following figure serves to illustrate the difference between the terms pipelined and overlapping.



FU type 3 pipelined

execution time: 3 (3 stages)
instr. rate: 1
overlapping factor: none



FU type 5 with overlapping

execution time: variable
instr. rate: 1
overlapping factor: 3 (3 input register)

In the FU type 3, instructions are processed in a pipelined fashion. The unit guarantees a result rate of one result per clock tick. This empties the pipeline as fast as it can be filled with instructions. None of the processing stages can be stopped for synchronization; only the whole FU can be frozen by disabling the clock signal of the FU.

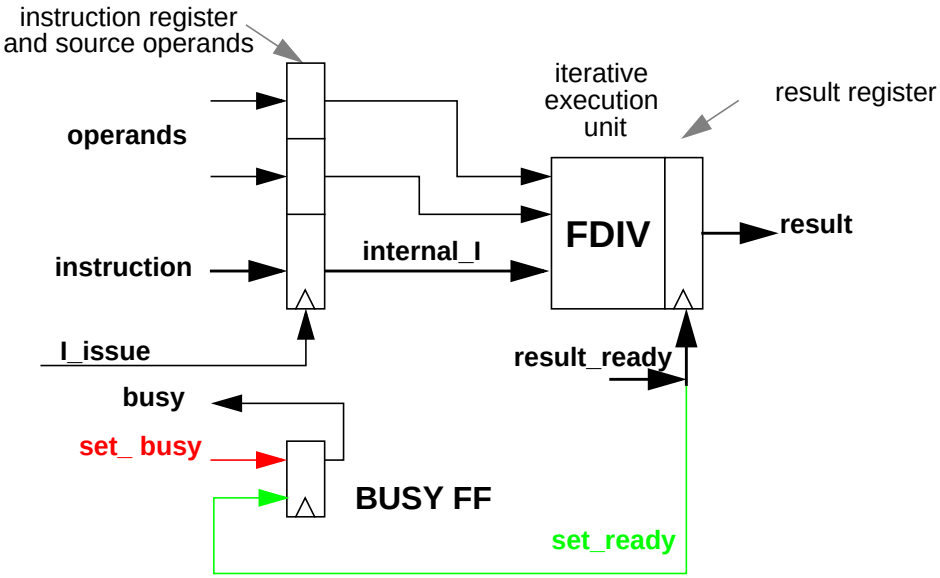
The FU type 5 has a synchronization input in stage f2 of the processing stages. Assuming the synchronization input is not ready, processing by f2 must stop immediately. This does not necessarily stop the instruction input to the instruction input queue. A load-store unit featuring reservation stations is good example of such FU. Only a full queue stops instruction issue, and this event halts the whole CPU. FU type 5 can be used to model a load-store unit that synchronizes to external memory control and halts the CPU only in the case where the number of instructions exceeds the overlapping factor.

An example for FU of type 4 is the iterative floating point division unit.

Superscalar Processors

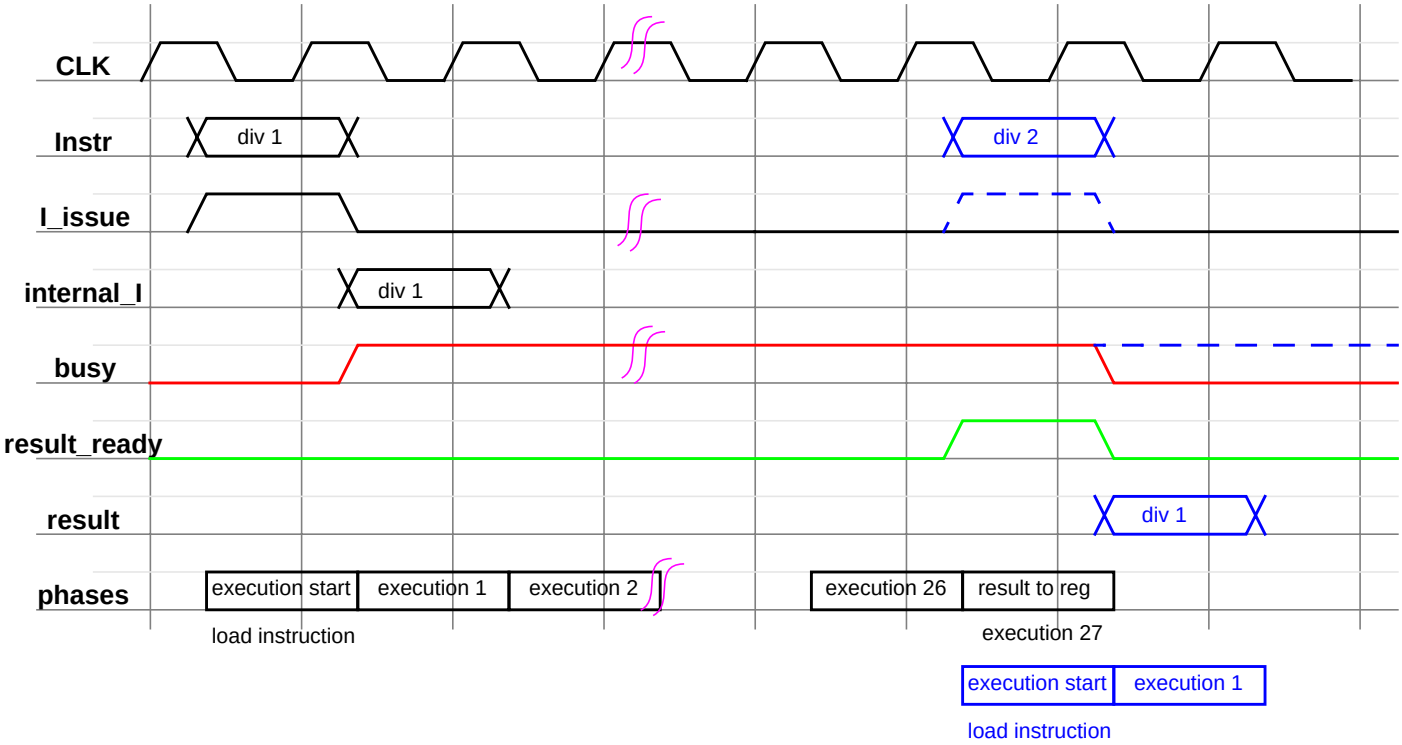
FU Synchronization: Example of FDIV-Unit

When an instruction is issued to an iterative execution unit, this unit requires a number of clocks to perform the function (e.g. 27 clocks for fdiv). During this time, the FU is marked as "busy" and no further instruction can be issued to this FU. When the result becomes available, the result register must be clocked and the FU becomes "ready" again.



Eq. for busy check:

```
I_issue = I_for _div_FU & ( /busy + busy & resul_ ready )
```



A new instruction can be issued to the unit at the same clock edge when the internal result is transferred to the result register (advancement of data in the pipeline stages!). The instruction issue logic checks not only the busy signal but also the result_ready signal and can thus determine the just going ready ("not busy") FU.

PowerPC 620

Overview of the 620 Processor

The 620 Processor is the first 64-bit Implementation of the PowerPC-Architecture.

- 200 (300) MHz clock frequency
- 7 million transistor count, power dissipation < 17-23 W @ 150MHz, 3.3V
- 0.5 µm CMOS technology, 625 pin BGA
- max. 400 MFLOPS floating-point performance (dp)
- 800 MIPS peak instruction execution
- four instructions issued per clock, instruction dispatch in order
- out-of-order execution, in-order completion
- multiple independent functional units, int, ld/st, branch, fp
- reservation stations, register renaming with rename buffers
- five stages of master instruction pipeline
- full instruction hazard detection
- separate register files for integer (GPR) and floating point (FPR) data types
- branch prediction with speculative execution
- static and dynamic branch prediction and branch target instruction buffer
- 4 pending predicted branches
- 32k data and 32k instruction cache, 8 set associative, physically addressed
- non-blocking load access, 64-bytes cache block size
- separate address ports for snooping and CPU acesses
- level-2 cache interface to fast SSRAM (1MB - 128MB)
- MMU, 64-bit effective address, 40-bit physical address
- 64-entry fully associative ATC
- multiprocessor support, bus snooping for cache coherency (MESI)
- pipelined snoop response
- dynamic reordering of loads and stores
- explicit address and data bus tagging, split-read protocol
- 128-bit data bus crossbar compatible, cpu id tagged
- bus clock = processor clock/n (n = 2,3,4)

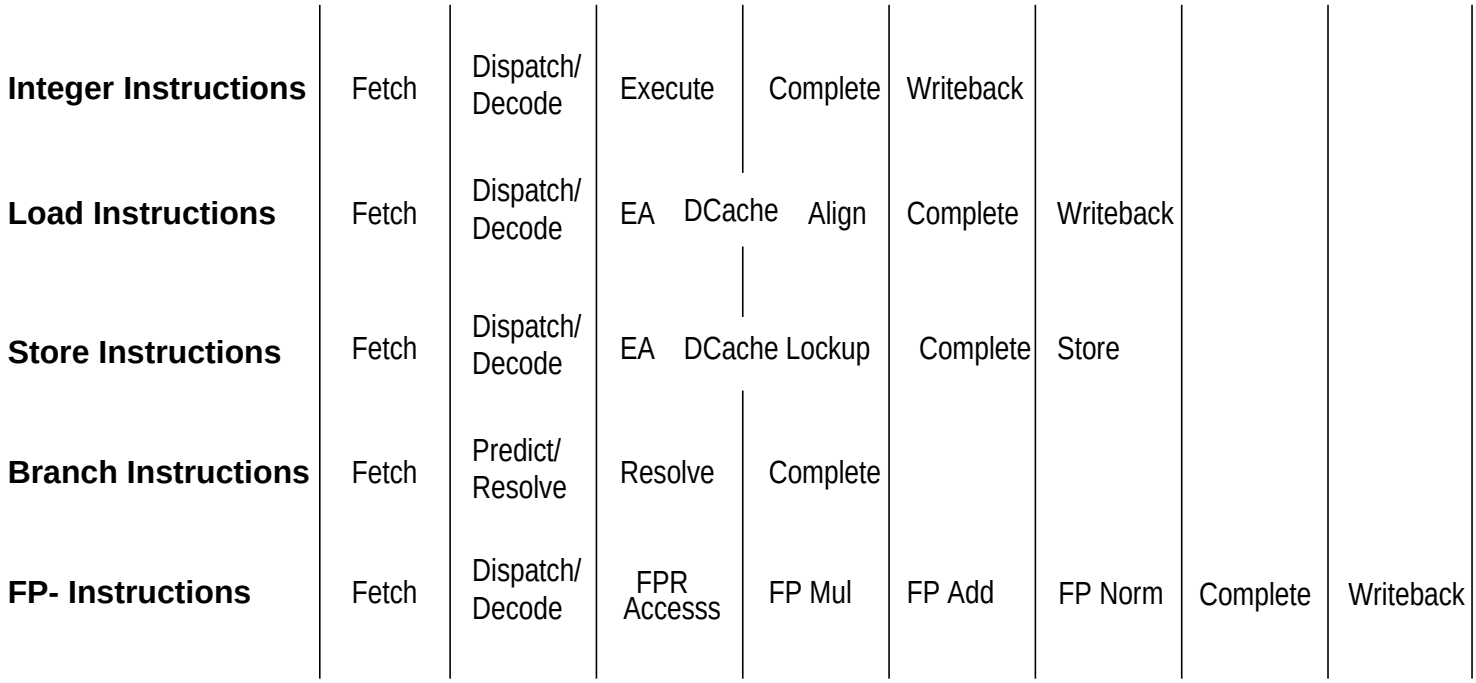
PowerPC 620

Pipeline Structure

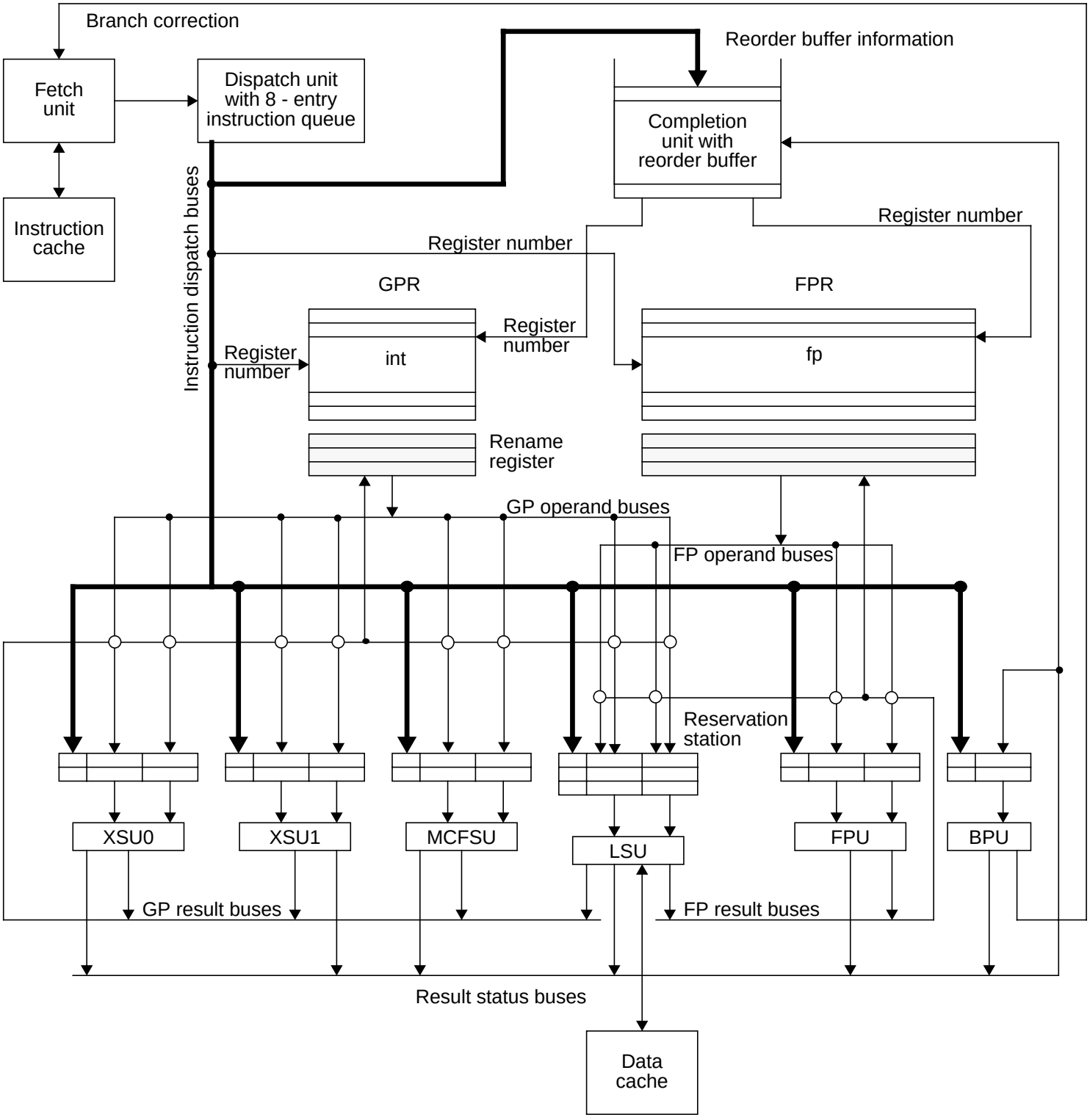
The 620 master instruction pipeline has five stages. Each instruction executed by the processor will flow through at least these stages. Some instructions flow through additional pipeline stages.

The five basic stages are:

- Fetch
- Dispatch
- Execute
- Complete "in order completion"
- Writeback



PowerPC 620


Branch correction
Reorder buffer information
Fetch unit
Dispatch unit with 8 - entry instruction queue
Completion unit with reorder buffer
Instruction cache
Instruction dispatch buses
Register number
Register number
GPR
FPR
Register number
Register number
int
fp
Rename register
GP operand buses
FP operand buses
Reservation station
XSU0
XSU1
MCFSU
LSU
FPU
BPU
GP result buses
FP result buses
Result status buses
Data cache

PowerPC 620

Data Cache Management Instructions

A very interesting feature of the PowerPC 620 is the availability of five-user-mode instructions which allow the user some explicit control over the caching of data. Similar mechanisms and a compiler implementation are proposed in [Lam, M.; Software Pipelining: An Effictive Scheduling Technique for VLIW Machines; in: Proc. SIGPLAN '88, Conf. on Programming Language Design and Implementation, Jun. 1988, pp. 318-328].To improve the cache hit rate by compiler-generated instructions, five instructions are implemented in the CPU, which control data allocation and data write-back in the cache.

- **DCBT - Data Cache Block Touch**
- **DCBTST - Data Cache Block Touch for Store**
- **DCBZ - Data Cache Block Zero**
- **DCBST - Data Cache Block Store**
- **DCBF - Data Cache Block Flush**

- **EIEIO - Enforce in-Order Execution of I/O**

A DCBT is treated like a load without moving data to a register and is a programming hint intended to bring the addressed block into the data cache for loading. When the block is loaded, it is marked shared or exclusive in all cache levels. It allows the load of a cache line to be scheduled ahead of actual use and can hide the latency of the memory system into useful computation. This reduces the stalls due to long latency external accesses.

A DCBTST is treated like a load without moving data to a register and is a program ming hint intended to bring the addressed block into the data cache for storing. When the block is loaded, it is marked modified in all cache levels. This can be helpful if it is known in advance that the whole cache line is overwritten, as it usually is in stack accesses.

If the storage mode is write-back cache-enabled then the addressed block is established in the data cache by the DCBZ without fetching the block form main storage, and all bytes of the block are set to zero. The block is marked modified in all cache levels. This instruction supports for example an OS creating new pages initialized to zero.

The DCBST, or data cache block clean, will write modified cache data to a memory and leave the final cache state marked shared or exclusive. Treated like a load with respect to address translation and protection.

The DCBF, or data cache block flush, instruction is defined to write modified cache data to the memory and leave the final cache state marked invalid.

PowerPC 620

Storage Synchronization Instructions

The 620 Processor supports atomic operations (read/modify/write) with a pair of instructions. The *Load and Reserve* (LWARX) and the *Store Conditional* (STCX) instructions form a conditional sequence and provides the effect of an atomic RMW cycle, but not with a single atomic instruction. The conditional sequence begins with a *Load and Reserve* instruction; may be followed by memory accesses and/or computation that include neither a *Load and Reserve* nor a *Store Conditional* instruction; and ends with a *Store Conditional* instruction with the same target address as the initial *Load and Reserve*.

These instructions can be used to emulate various synchronization primitives, and to provide more complex forms of synchronization.

The reservation (it exists only one per processor!!) is set by the lwarx instruction for the address EA. The conditionality of the *Store Conditional* instruction's store is based only on whether a reservation exists, not on a match between the address associated with the reservation and the address computed form the EA of the *Store Conditional* instruction. If the store operation was successful the CR field is set and can be tested by a conditional branch.

A reservation is cleared if any of the following events occurs:

- The processor holding the reservation executes another *Load and Reserve* instruction; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes a *Store Conditional* instruction to **any** address.
- Another processor or bus master device executes any Store instruction to the address associated with the reservation.

The reservation granule for the 620 is 1 aligned cache block (64 bytes).

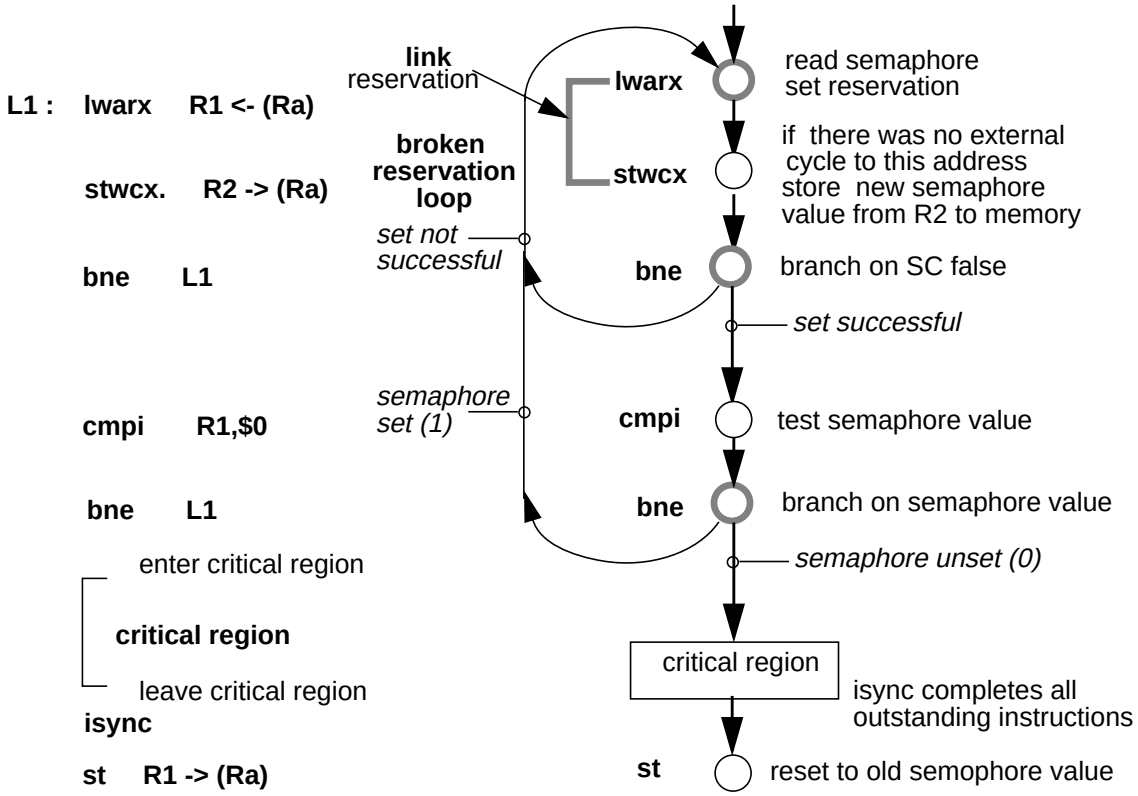
Due to the ability to reorder load and store instruction in the bus interface unit (BIU) the sync instruction must be used to ensure that the results of all stores into a data structure, performed in in a critical section of a program, are seen by other processors before the data structure is seen as unlocked.

PowerPC 620

Storage Synchronization Instructions

This implementation is a clever solution, because the external bus system is not blocked for the test-and-set sequence. All other bus masters can continue using the bus and the memory system for other memory accesses. The probability that another processor is accessing the same address in between lwarx and stwcx is very low, because of the small number of instructions used for the modify phase (number depends on the synchronization primitive).

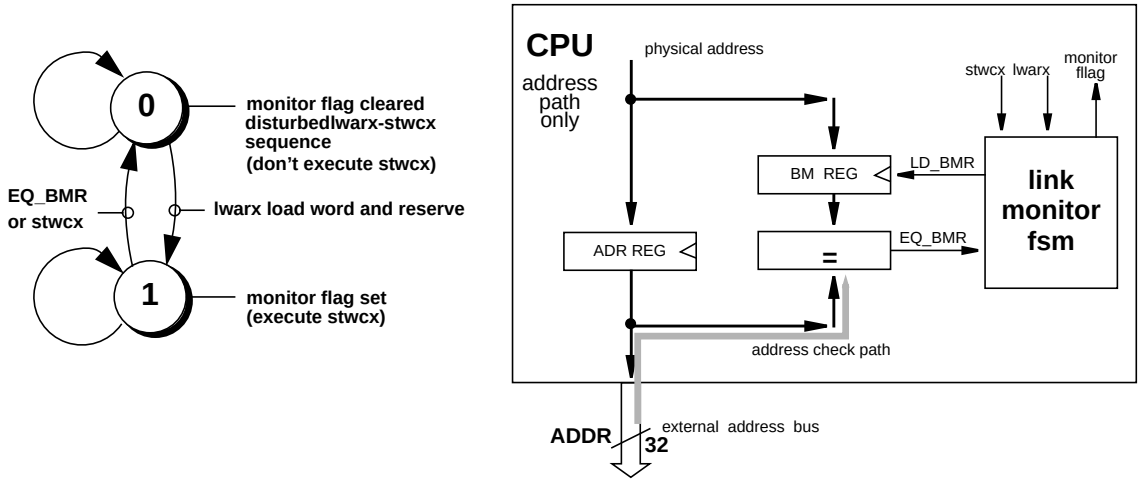
If the lwarx instruction finds a semaphore which was already set by another processor or process, the next access of the semaphore within the reservation loop is serviced from the cache, so that no external bus activity is performed by the busy waiting test loop. The release of this semaphore shows up in the cache by the coherency snoop logic, which usually invalidates (or updates) the cache line. The next lwarx gets the released semaphore and tries to set it with the stwcx instruction. The execution of instructions from the critical region can be started after testing the monitor_flag by the bne instruction. Each critical region must be locked by such a binary semaphore or by a higher-level construct. The semaphore is released by storing a "0" to the semaphore address. The coherence protocol guarantees that this store is presented on the bus, although the cache may be in copy-back mode. The following instruction sequence implements a binary semaphore sequence (fetch-and-store). It is assumed that the address of the semaphore is in Ra, the value to set the semaphore in R2.



PowerPC 620

Storage Synchronization Instructions

Hardware resources for the supervision of the reservation.



These functional components are mapped to the on-chip cache functions. This causes the reservation granule to be one cache line. The snooping of the cache provides the address comparision and the address tag entry contains the address of the reservation.

The following instruction sequence implements an optimized version of the **binary** semaphore instruction sequence, which increases the propability to run an undisturbed lwarx-stwcx sequence.

L1 :	**ld R1 <- (Ra)**	read semaphore using normal load into R1
	cmp R1 , R2	compare semaphore value to R2
	beq L1	loop back if semaphore not free (1)
atomic by reservation	**lwarx R1 <- (Ra)**	set the reservation to start the atomic operation (1) when there is a high propabiltity to succeed
	stwcx. R2 -> (Ra)	store conditional semaphore from R2 if there was no external cycle to this address
	bne L1	branch back if reservation failed (likely not taken)
	cmp R1 , $0	test semaphore value again
	bne L1	branch back on 'semaphore set' due to intermediate access from another processor
	critical region	
	st R0 = 0 -> (Ra)	reset semaphore value (0)

It is assumee that the address of the semaphore is in Ra, the value to compare with in R2

Synchronization

Definition : ***Synchronization is the enforcement of a defined logical order between events. This establishes a defined time-relation between distinct places, thus defining their behaviour in time.***

Definition : ***A process is a unit of activity which executes programs or parts thereof in a strictly sequential manner, requiring exactly one processor [Gil93]. It consists of the program code, the associated data, an address space, and the actual internal state.***

Definition : ***A thread or lightweight process is a strictly sequential thread of control (program code) like a little mini-process. It consists of a part of the program code, featuring only one entry and one exit point, the associated local data and the actual internal state. In contrast to a process, a thread shares its address space with other threads [Tan92].***

Es gibt zwei unterschiedliche Situationen, die eine Synchronisation zwischen Prozessen erfordern.

- - die Benutzung von shared resources und shared data structures.
  - Die Prozesse müssen in der Lage sein die gemeinsamen Resourcen zu beanspruchen, und ohne Beeinflussung voneinander benutzen zu können.
  - mutual exclusion

- - die Zusammenarbeit von Prozessen bei der Abarbeitung von einer Task
  - Die Prozesse müssen bei der Abarbeitung der Teilaufgaben in eine zeitlich korrekte Reihenfolge gebracht werden. Die Datenabhängigkeit zwischen Prozessen die Daten erzeugen (producer) und denen die Daten verbrauchen (consumer) muß gelöst werden
  - process synchronisation -> RA-2

Mutual exclusion

Ein bestimmtes Objekt kann zu jedem Zeitpunkt höchstens von einem Prozeß okkupiert sein.

Zur Veranschaulichung der Problematik soll folgendes Beispiel dienen.

Mehrere Prozesse benutzen einen Ausgabe-Kanal. Die Anzahl der ausgegebenen Daten soll in der Variable count gezählt werden.

Dazu enthält jeder Prozeß die Anweisung
count := count + 1
nach seiner Datenausgabe.

Synchronization

Atomic operations

sequential consistency-

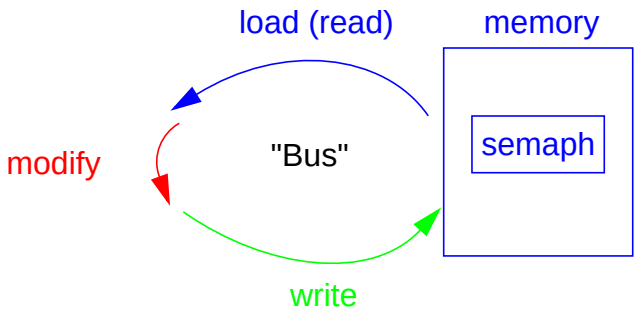
requires that all instructions issued by each processor appear to execute in the same order when observed by all other processors.

Reading of instructions?

requires a mechanism for implementation

synchronization instructions

read - modify - write



Atomic
Zugriff verhindern
- Bus reserviert P O
  gesamter Speicher blockiert
- Cacheline snooping
  Zugriff auf diese Adresse verh. (?)

Synchronization

Diese Anweisung wird in folgende Maschinenanweisungen übersetzt .

```
ld count , R1
add R1 , #1
store R1 , count
```

Diese Sequenz befindet sich in jedem der Prozesse, welche parallel ihre Instruktionen abarbeiten. Dadurch könnte folgende Sequenz von Instruktionen entstehen.

```
P1: ld    count , R1
P2: ld    count , R1
P2: add   R1 , #1
P2: store R1 , count
P1: add   R1 , 1
P1: store R1 , count
```

Damit ist count nur um 1 erhöht worden, obwohl die Instruktionssequenz zweimal durchlaufen wurde und count um 2 erhöht sein sollte.

Um dieses Problem zu vermeiden führt man den Begriff des kritischen Bereichs (critical region) ein.

Eine critical region ist eine Reihenfolge von Anweisungen, die unmittelbar, d.h. ohne Unterbrechung, von einem Prozeß ausgeführt wird. Der Eintritt in einen solchen kritischen Bereich wird nur einem Prozeß gestattet.

Andere Prozesse, die diesen Bereich ebenfalls benutzen wollen, müssen warten bis der belegende Prozeß den Bereich wieder verläßt und damit zur Benutzung freigibt.

Möglichkeiten zur Realisierung der critical region:

- Software-Lösung (zu komplex !)
- Interrupt-Abschaltung (single processor)
- binary semaphores
- einfache kritische Bereiche
- Monitors

Synchronization

Interrupt - Abschaltung

Ist in einem System nur ein Prozessor vorhanden, der die Prozesse ausführt, so kann man die mutual exclusion durch das Abschalten der Interrupts erreichen. Sie sind die einzige Quelle, durch die eine Instruktionssequenz unterbrochen werden kann

> Praktische Ausführung : der kritische Bereich wird als Interrupt-Handler geschrieben. Eintritt durch Auslösen eines Traps. Interrupts sperren. Kritischen Bereich bearbeiten. Interrupt freigeben. Rücksprung zum Trap.

Der Basismechanismus zur Synchronisation ist die unteilbare Operation (atomic operation)

Unteilbare Operationen (ATOMIC OPERATIONS)

Die unteilbare Operation zur Manipulation einer gemeinsamen Variablen bildet die Grundlage für die korrekte Implementierung der mutual exclusion

Goal: generating a logical sequence of operations of different

- threads
- processes
- processors

Implementation: Atomic operations

- read-modify-write
- test-and-set (machine instruction)
- load-incr-store
- load-decr-store
- loadwith-store conditional
- reservation
- lock/unlock
- fetch-and-add

Synchronization

Multiprocessor Systems using a bus as Processor-Memory interconnect network can use a very simple mechanism to guarantee atomic (undividable) operations on semaphore variables residing in shared memory.

The bus as the only path to the memory can be blocked for the time of a read-modify-write (RMW) sequence against intervening transactions from other processors. The CAS2 Instruction of the MC68020 is an example of such a simple mechanism to implement the test-and-set sequence as an non-interruptable machine instruction. Sequence of operations of CAS2 instruction

1. read of semaphore value from Mem(Ra) into register Rtemp
2. compare value in Rtemp to register Rb
3. conditional store of new sempahore to Mem(Ra) from register Rc

Synchronization

Undividable operations

lock — begin interlocked sequence

Causes the next data load or store that appears on the bus to assert the LOCK # signal, directing the external system to lock that location by preventing locked reads, locked writes, and unlocked writes to it from other processors. External interrupts are disabled from the first instruction after the lock until the location is unlocked.

unlock — end interlocked sequence

The next load or store (regardless of whether it hits in the cache) deasserts the LOCK # signal, directing the external system to unlock the location, interrupts are enabled when the load or store executes.

These instructions allow programs running either user or supervisor mode to perform atomic read-modify-write sequences in multiprocessor and multithread systems. The lock protocol requires the following sequence of activities:

- lock
- Any load instruction that happens on the bus starts the atomic cycle. This load does not have to miss the cache; it is forced to the bus.
- unlock
- Any store instruction terminates the atomic cycle.

There may be other instructions between any of these steps. The bus is locked after step 2 and remains locked until step 4. Step 4 must follow step 1 by 30 instructions or less: otherwise an instruction trap occurs.

The sequence must be restartable from the **lock** instruction in case a trap occurs. Simple read-modify-write sequences are automatically restartable. For sequences with more than one store, the software must ensure that no traps occur after the first non-reexecutable store.

Synchronization

Starting with the memory read operation the RMW_ signal from the CPU tells the external arbitration logic not to rearbitrate the bus to any other CPU. Completing the RMW-instruction by the write to memory releases the RMW_ signal permitting general access to the bus (and to the memory).

- test-and-set (two paired machine instructions)

These two instructions are paired together to form the atomic operation in the sense, that if the second instruction (the store conditional) was successful, the sequence was atomic. The processor bus is not blocked between the two instructions. The address of the semaphore is supervised by the "reservation", set by the first instruction (load-and-reserve). [see also PowerPC620 Storage Synchronization]

Synchronization

binary Semaphores

Definition

Die **binäre Semaphore** ist eine Variable, die nur die zwei Werte ’0’ oder ’1’ annehmen kann. Es gibt nur die zwei Operationen P und V auf dieser Variablen.

’0’ entspricht hier einer gesetzten Semaphore, die den Eintritt in den kritischen Bereich verbietet

1. die P-Operation

   der Wert der Variablen wird getestet

   - ist er ’1’, so wird er auf ’0’ gesetzt und die kritische Region kann betreten werden.
   - ist er ’0’, so wird der Prozess in eine Queue eingereiht, die alle Prozesse enthält, die auf dieses Ereignis (binäre Semaphore = ’1’) warten.

2. die V-Operation

   die Queue wird getestet

   - ist ein Prozess in der Queue, wird er gestartet. Es wird bei mehreren Prozessen genau einer ausgewählt (z.B. der erste)
   - ist die Queue leer, wird die Variable auf ’1’ gesetzt

einfache kritische Bereiche

Um die korrekte Benutzung von Semaphoren zu erleichtern, wurde von Hoare der einfache kritische Bereich eingeführt:

```
with Resource do S
```

Die kritischen Statements S werden durch P und V Operationen geklammert und garantieren damit die mutual exclusion auf den in Resource zusammengefaßten Daten (shared Data).

Damit wird die Kontrolle über die Eintritts- und Austrittsstellen der mutual exclususion vom Compiler übernommen. Durch die Deklaration der shared variables in der Resource ist es dem Compiler auch möglich, Zugriffsverletzungen zu erkennen und damit die gemeinsamen Daten zu schützen.

Dadurch reduzieren sich die Fehler, die der Programmierer durch die alleinige Benutzung der P und V Operationen in Programme einbauen könnte, erheblich.

Synchronization

Monitore

Die Operationen auf shared data sind i.a. über das gesamte Programm verteilt. Diese Verteilung macht die Benutzung der gemeinsamen Datenstrukturen unübersichtlich und fehleranfällig. Faßt man die gemeinsamen Variablen und die auf ihnen möglichen Operationen in ein Konstrukt zusammen und stellt die mutual exclusion bei der Ausführung des Konstrukts sicher, so erhält man einen

monitor

oder auch secretary genannt.
The basic concepts of monitors are developed by C.A.R. Hoare.

Definition:

> Ein monitor ist eine Zusammenfassung von critical regions in einer Prozedur. Der Aufruf der Prozedur ist immer nur einem Prozess möglich.

Somit ist der Eintritt in den Monitor mit der P-Operation äquivalent und der Austritt mit der V-Operation.
Beispiel:

```
monitor MONITORNAME;
   /* declaration of local data */
   procedure PROCNAME(parameter list);
      begin
         /* procedure body */
      end
begin
   /* init of local data */
end
```

Java provides a few simple structures for synchronizing the activities of threads. They are all based on the concepts of monitors. A monitor is essentially a lock. The lock is attached to a resource that many threads may need to access, but that should be accessed by only one thread at a time.

The `synchronized` keyword marks places (variables) where a thread must acquire the lock before proceeding. [Patrick Niemeyer, Joshua Peck, Exploring JAVA, O'Reilly, 1996]

```
class Spreadsheet {
   int cellA1, cellA2, cellA3;

   synchronized int sumRow() {     // synchronized method
      return cellA1 + cellA2 + cellA3;
   }

   synchronized void setRow( int a1, int a2, int a3 ) {
      cellA1 = a1;
      cellA2 = a2;
      cellA3 = a3;
   }
...
}
```

In this example, synchronized methods are used to avoid race conditions on variables cellAx.