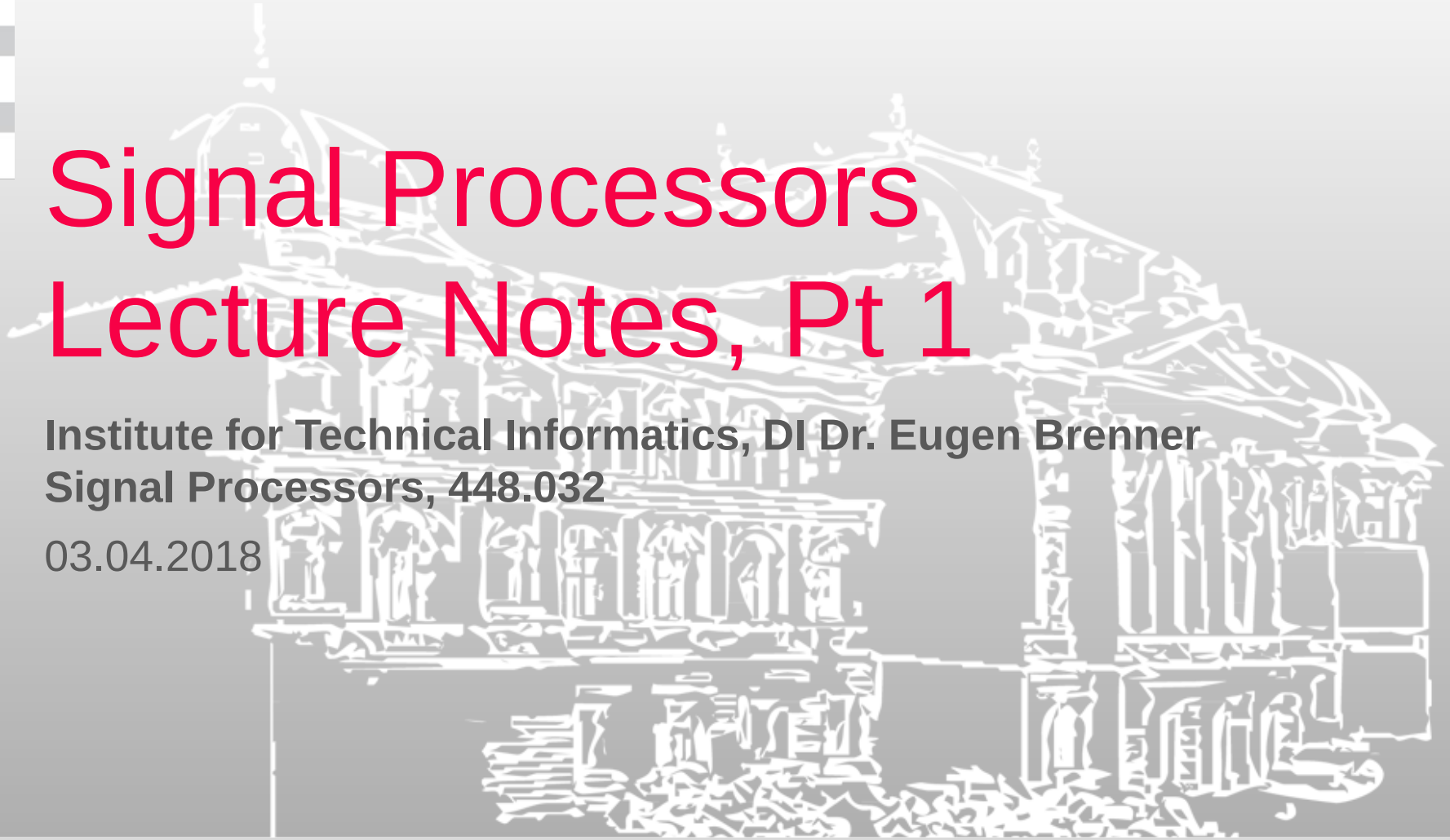
Signal Processors Lecture Notes, Pt 1

Institute for Technical Informatics, DI Dr. Eugen Brenner

Signal Processors, 448.032

03.04.2018



Chapter Overview

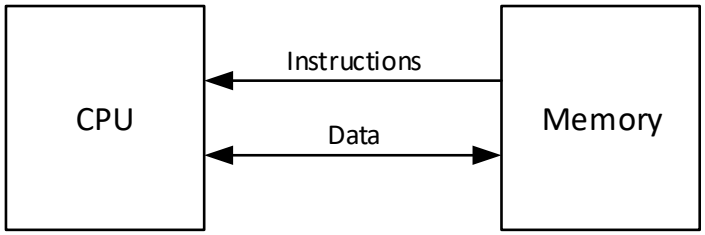
1. Microprocessor Fundamentals

- Processor & Programming Models
- Performance Criteria
- Data Path Implementation
- CISC vs. RISC
- Superscalar Processors
- Very Long Instruction Word Processors (VLIW)
- Detour: Programming Model / Program Execution
- Out-of-Order Execution
- Memory System
- Direct Memory Access
- Conclusion

Processor & Programming Models

Basic Model

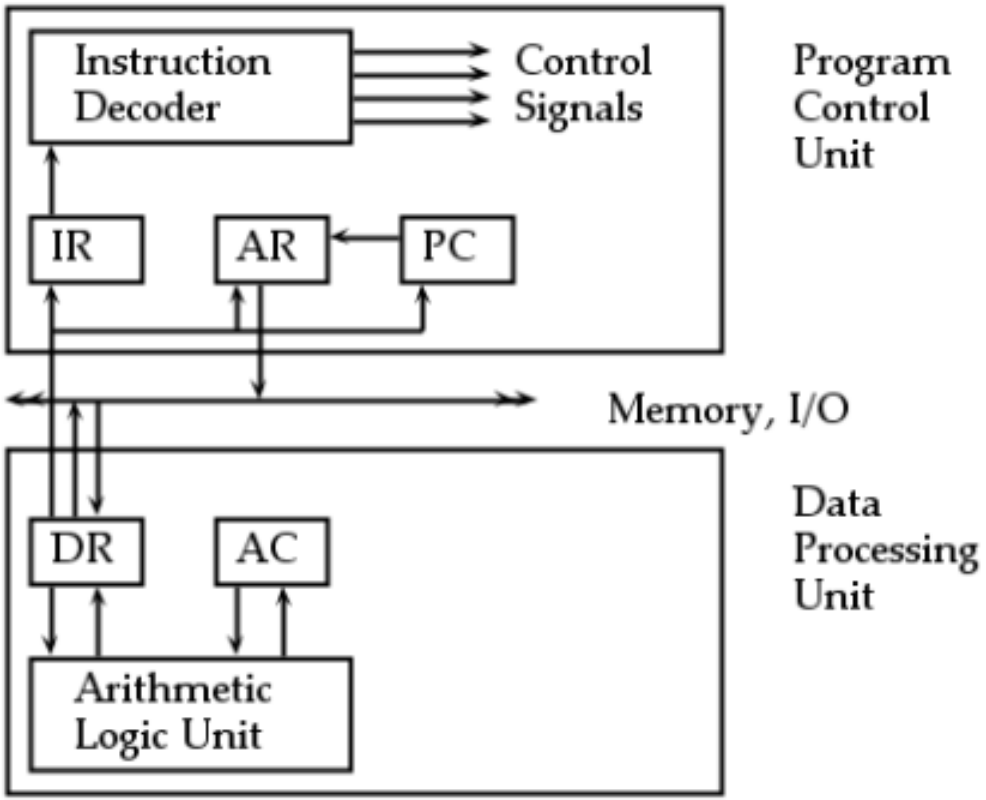
- Primary operation
  - “sequential” execution of instructions



- Program execution
  - CPU loads instruction and data if needed
  - CPU executes instructions sequentially
  - If required, stores results in register or memory

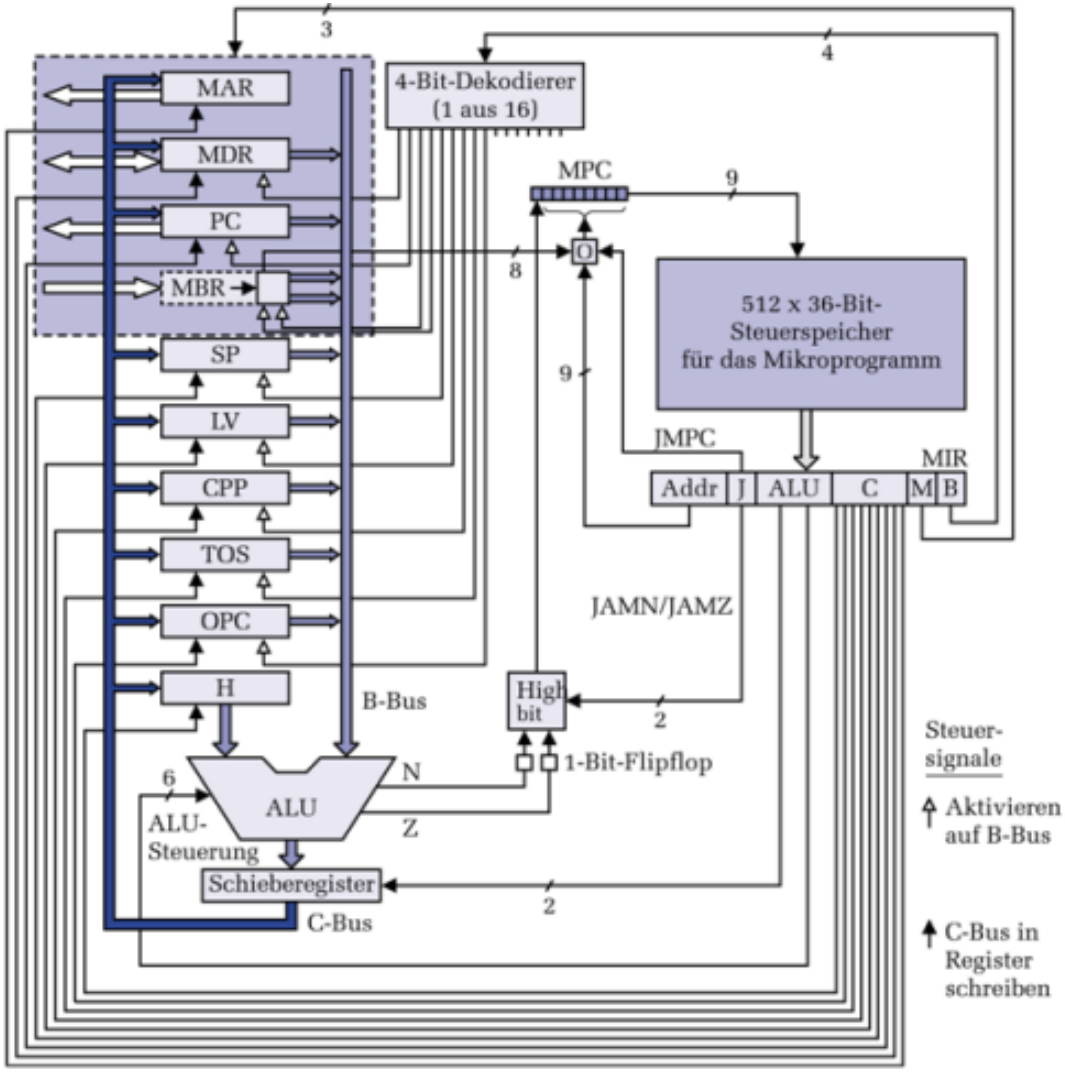
Processor Architecture: “Accu Model”

(Example)



Processor Internals: Microarchitecture

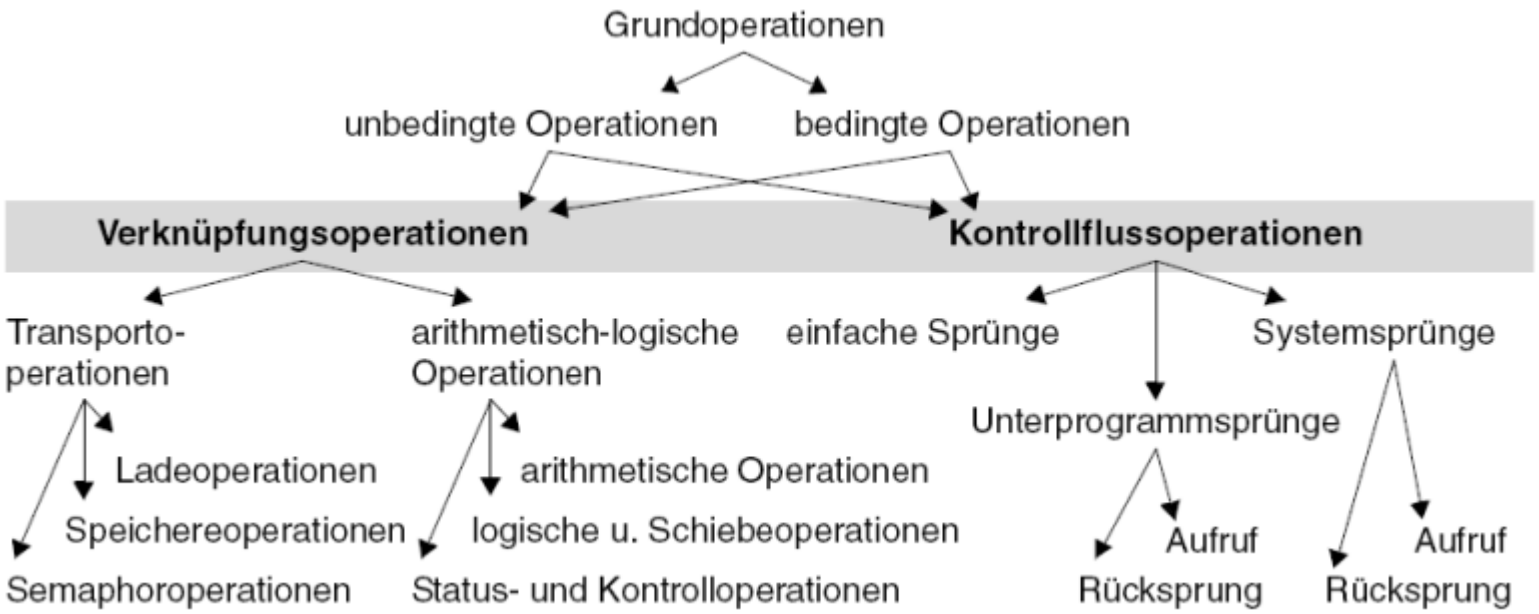
(Example)



Programming Model

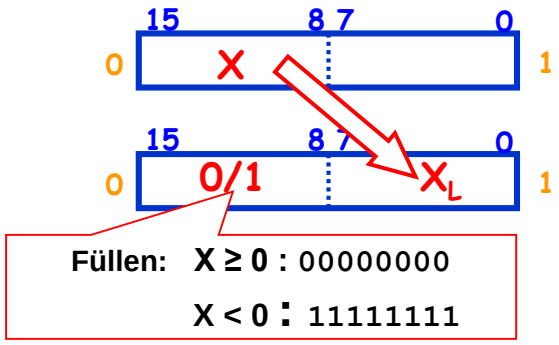
- Instruction set
  - Interface for the programmer/compiler
  - Set of “executable” instructions
  - Complexity
    - Instruction format
    - Data format
    - Available Registers
    - Addressing modes
- Instruction Set Architecture (ISA)
  - Defines “logical” behavior of a processor
  - Can be implemented by different processor
  - architectures resulting in *different performance*

Types of Instructions

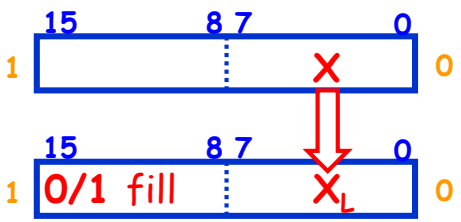


Big-endian vs. Little-endian

8-bit ⇨ 16-bit num.

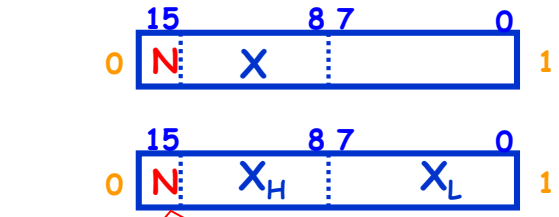


☹



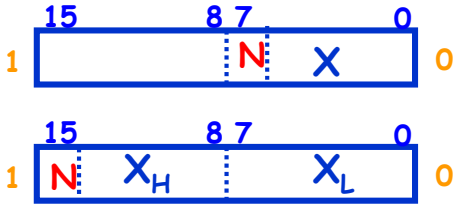
☺

Sign-Test of 8-bit and 16-bit numbers (N-Flag)



☺

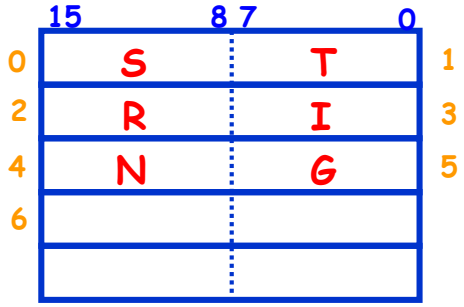
The same bit ! (in 16-bit-Register)



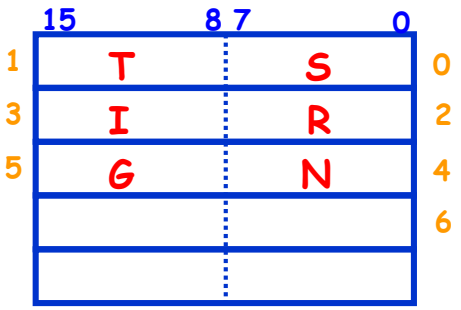
☹

Stringcomparison (sorting!)
Readability vs. Memory dumps

Blocktransfer



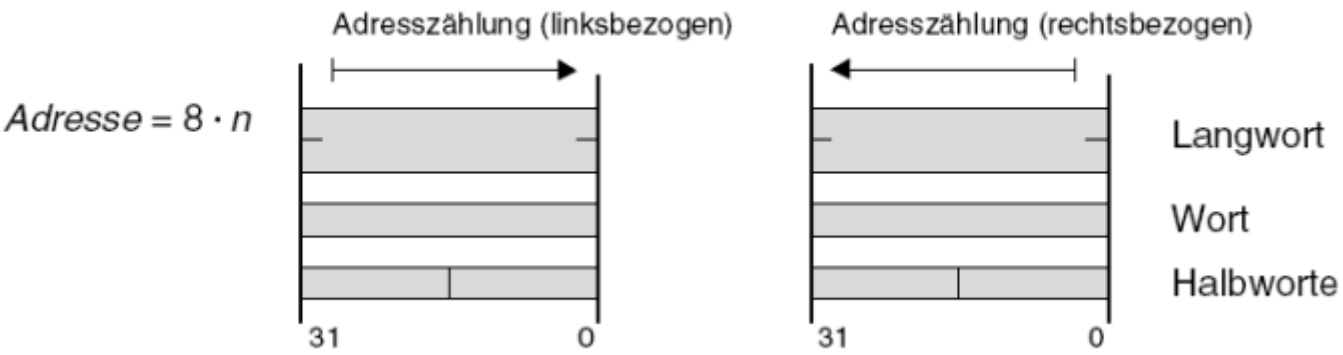
☺



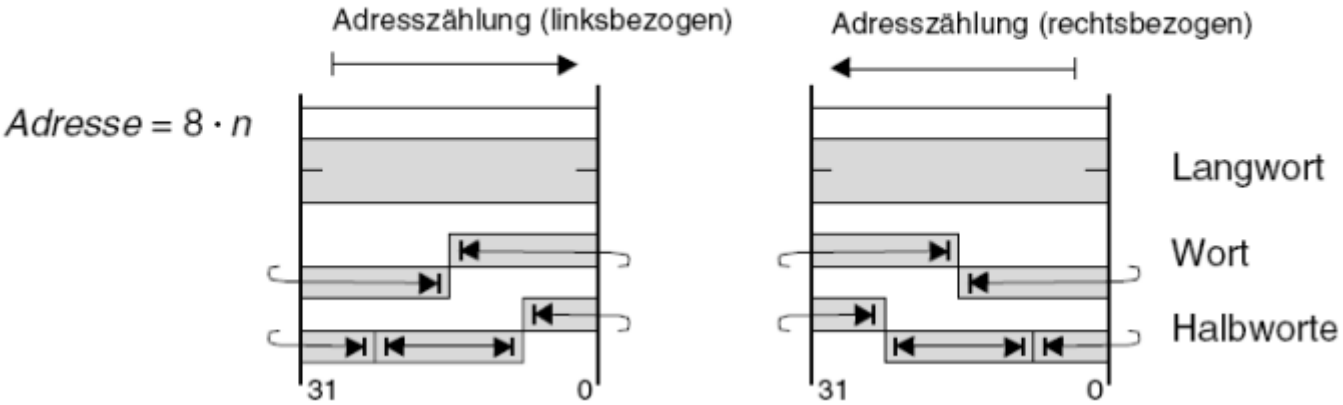
☹

Byte Order

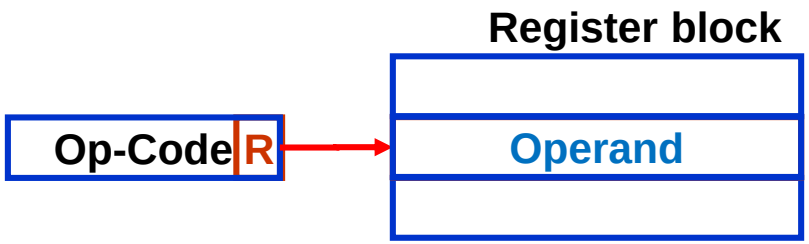
Ausgerichtete Daten (Aligned data)



Nicht ausgerichtete Daten (misaligned data)



Addressing Modes

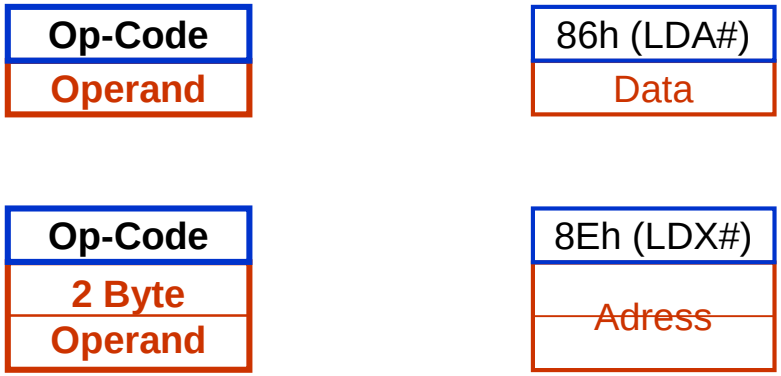


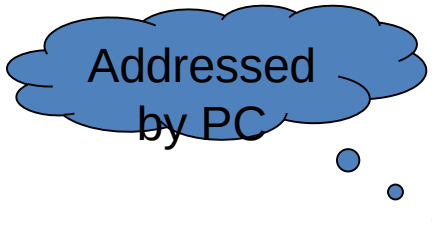
Inherent addressing / Register direct addressing

$R = \begin{cases} A \\ X \end{cases}$ e.g.: CLRA

Immediate addressing

Example:

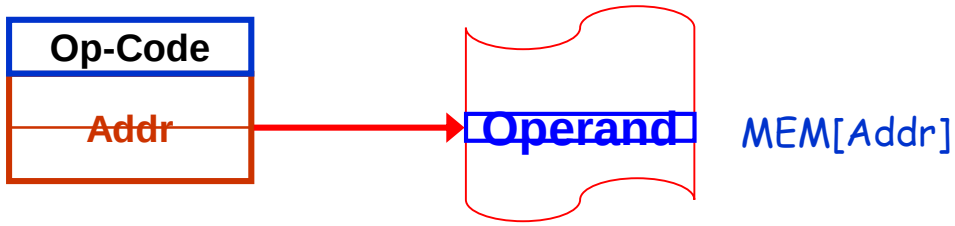




Operand is part of the opcode

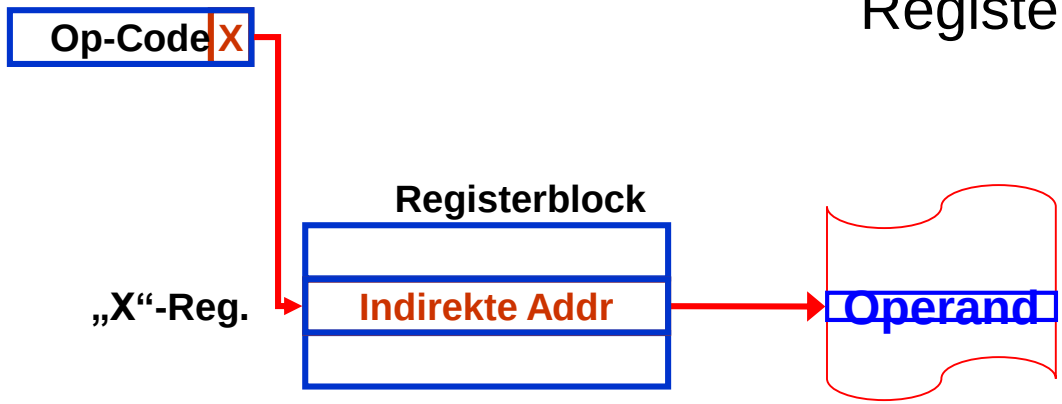
Addressing Modes

Absolute addressing



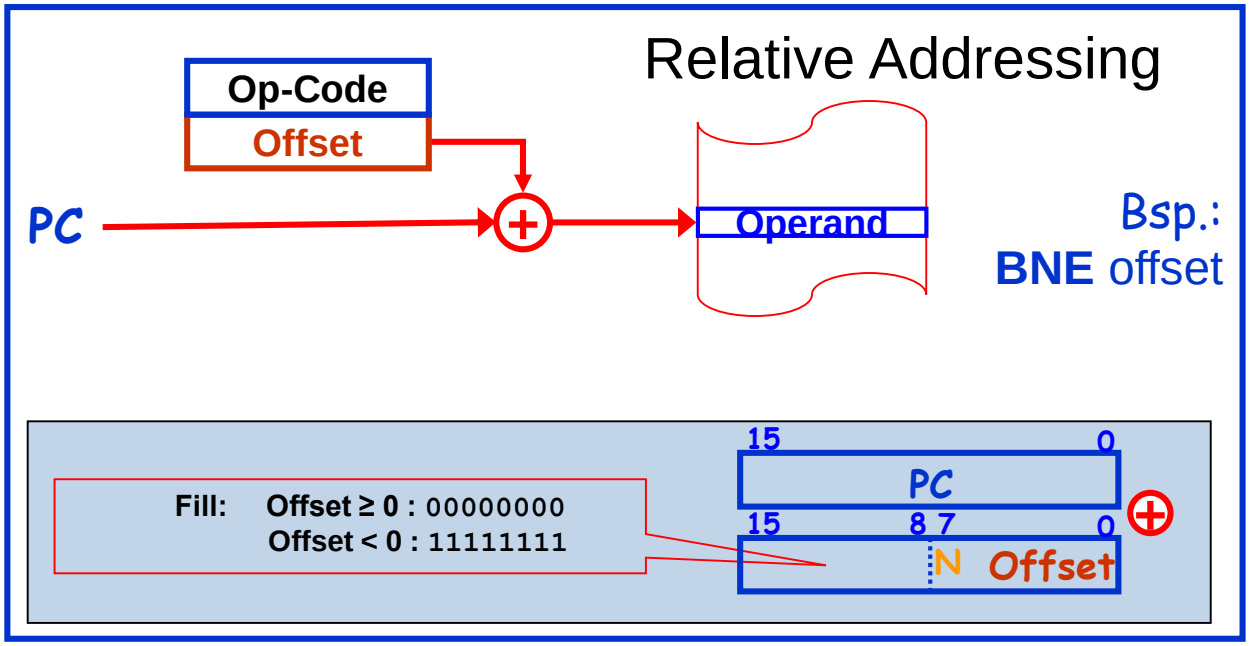
Bsp.:
LDA addr
STA addr

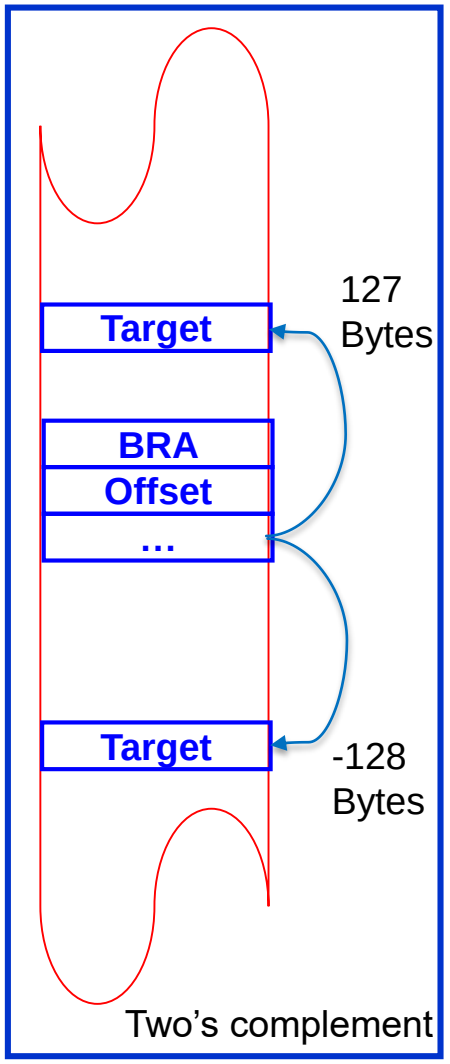
Register indirect addressing



Bsp.:
LDA @X
STA @X

Addressing Modes

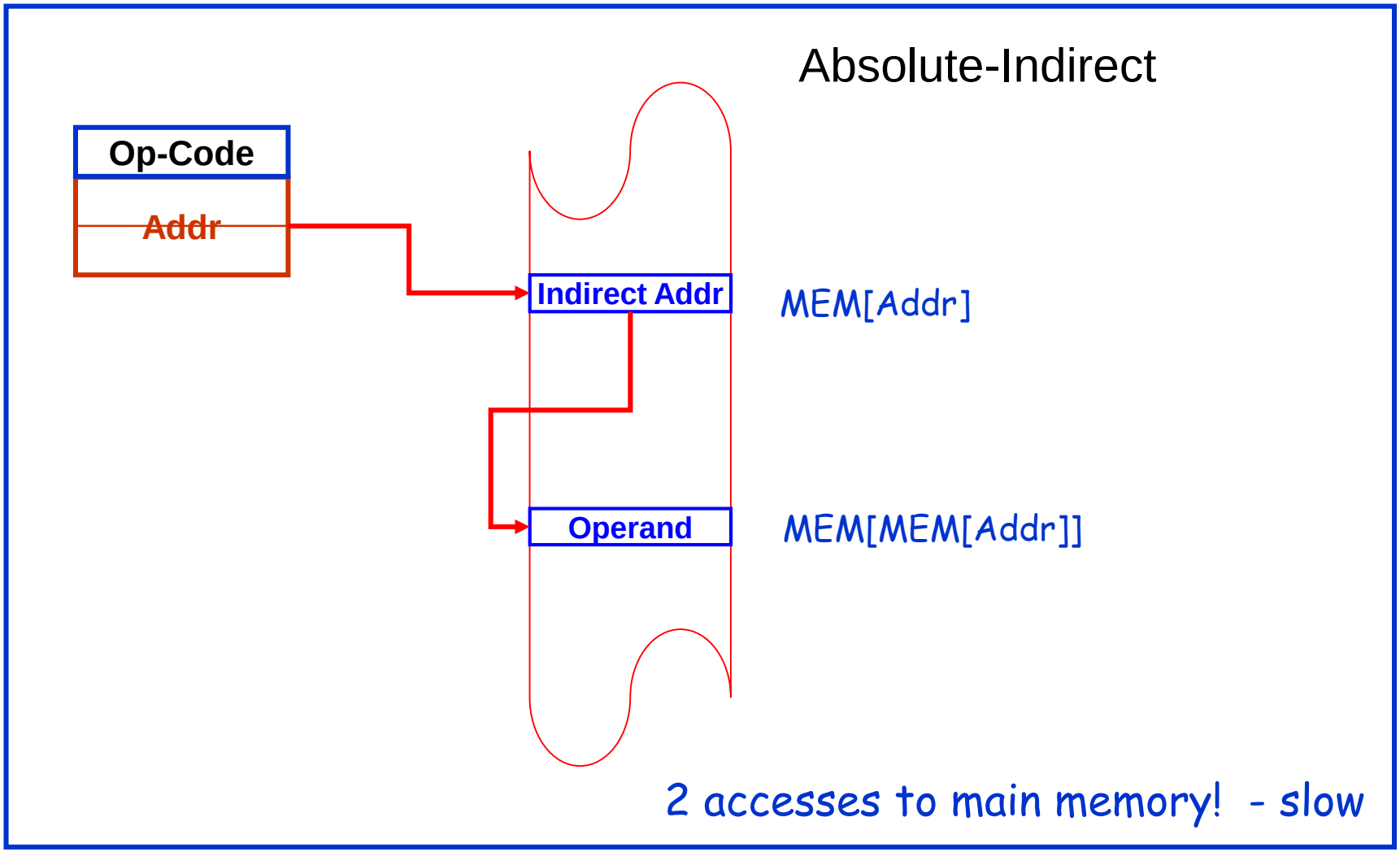




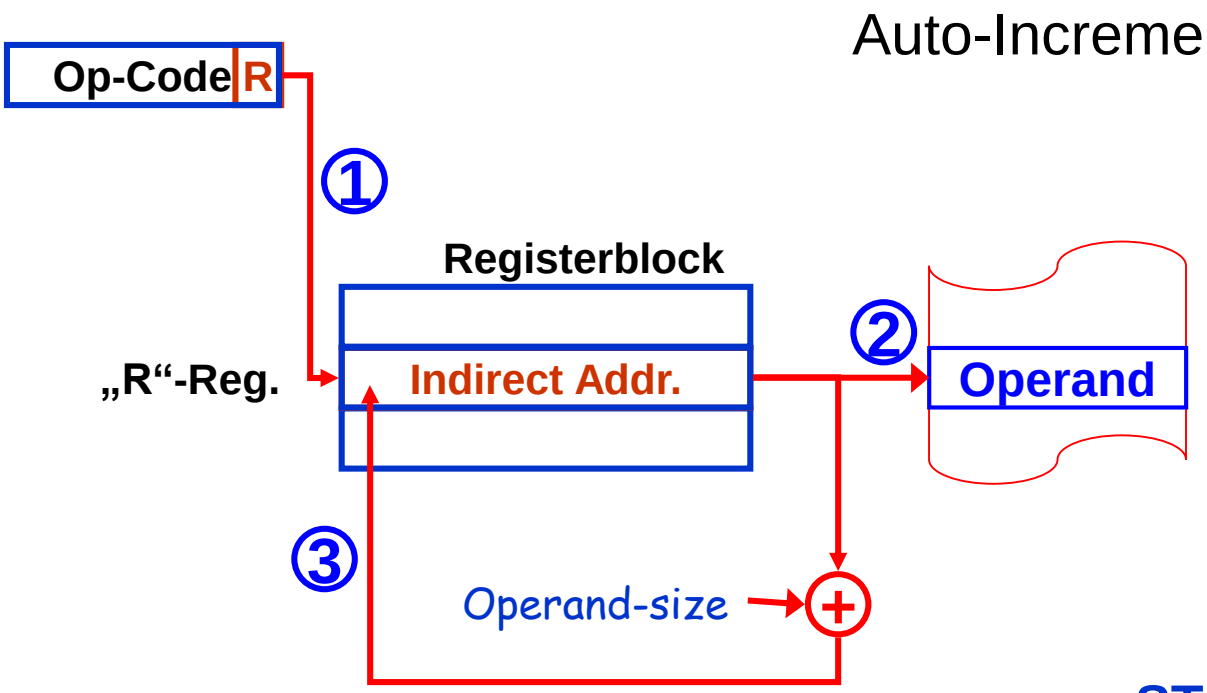
Usage: **„Position Independent Code“**

Programs can be executed everywhere in memory – because all of the addresses are relative to the program counter, no modifications are necessary

Addressing Modes



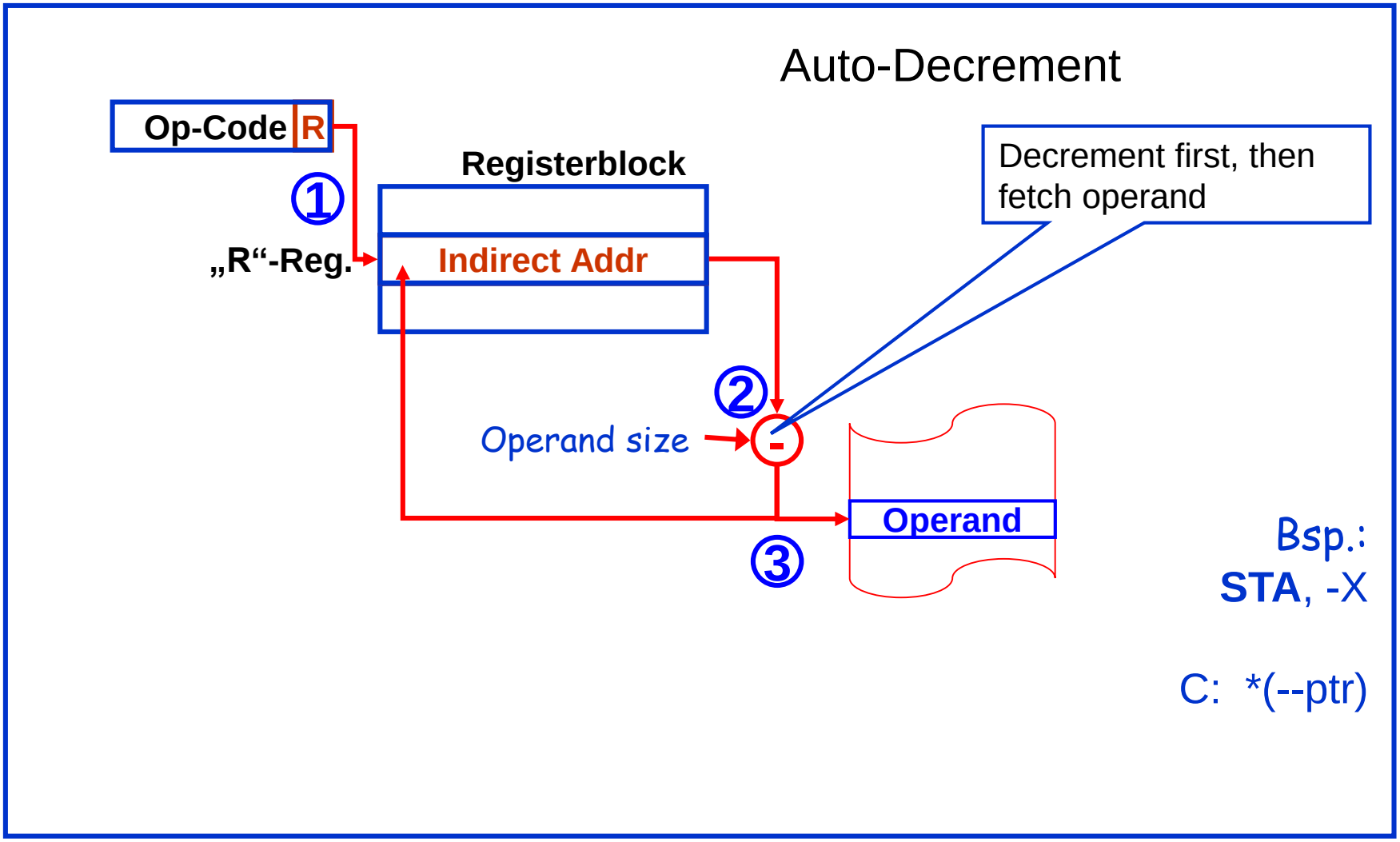
Addressing Modes



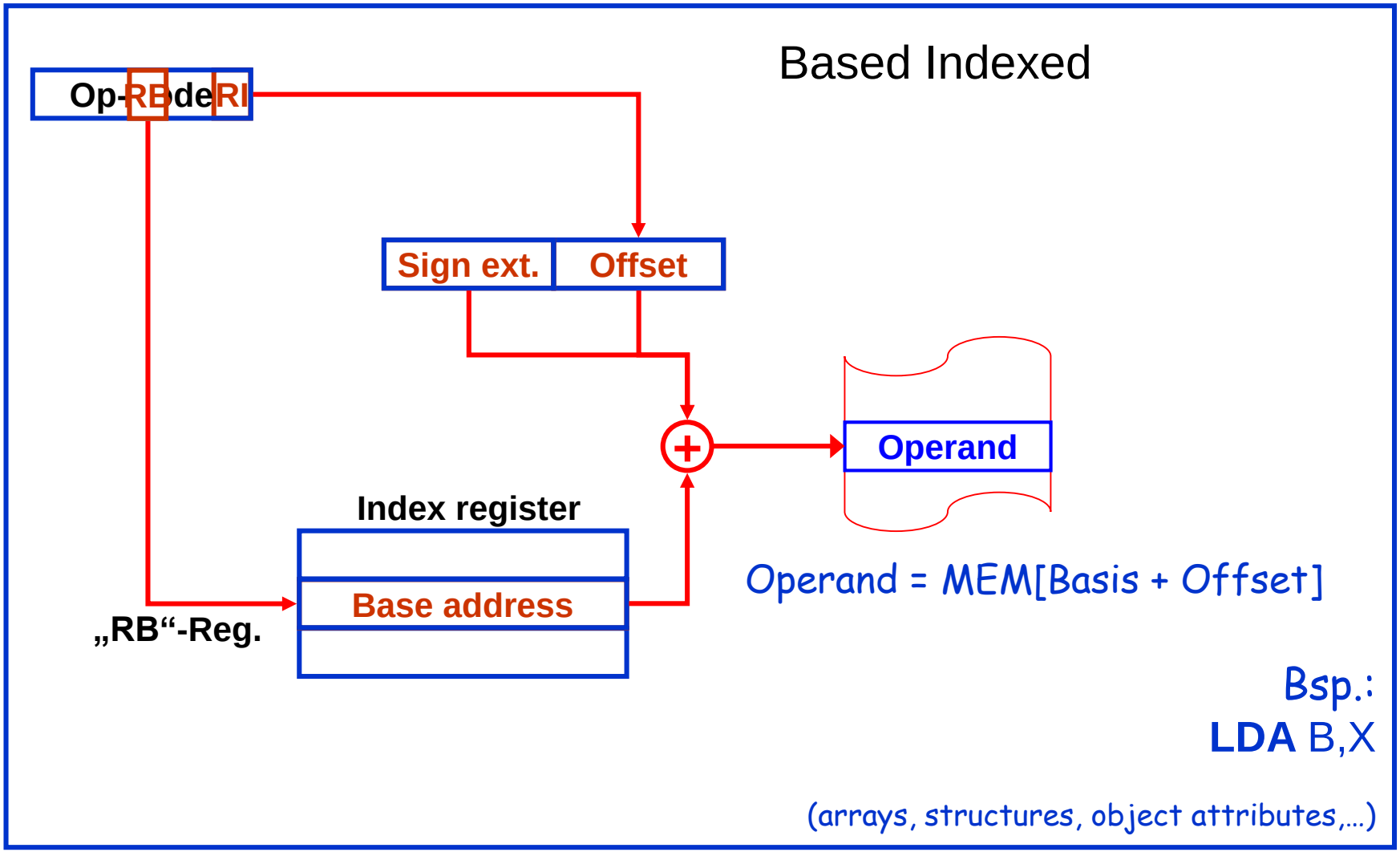
Bsp.:
STA, X+

C: *(ptr++)

Addressing Modes



Addressing Modes



Comparison of different Processors

Prozessor/Architektur (Hersteller)	Anzahl allgemeiner Register		Bitbreite			
	Gesamt	Direkt zugreifbar	Register-breite	Register-adressen	unmittelbare Operanden	Befehl
Alpha 21364 (Compaq) [25, 28]	32	32	64 Bit	5 Bit	8 Bit	32 Bit
Am29000 (AMD) [1]	192	192	32 Bit	8 Bit	8 Bit	32 Bit
ARM7TDMI (ARM) [10]	16	16	32 Bit	4 Bit	8 Bit	32 Bit[a]
Crusoe TM5800 (Transmeta) [187]	64	64	32 Bit	6 Bit	-[b]	-[b]
pa-8700 (HP) [90]	32	32	64 Bit	5 Bit	11 Bit	32 Bit
Itanium 2 (Intel, HP) [75, 78]	128	128	64 Bit	7 Bit	8 Bit[c]	41 Bit[d]
MC88100 (Motorola) [122]	32	32	32 Bit	5 Bit	16 Bit	32 Bit
MIPS64 20Kc (MIPS) [106]	32	32	64 Bit	5 Bit	16 Bit	32 Bit
Nemesis C (TU Berlin) [114, 198]	96	16	32 Bit	4 Bit	1 Bit	16 Bit
PowerPC 970 (IBM) [67]	32	32	64 Bit	5 Bit	16 Bit	32 Bit
UltraSPARC III Cu (Sun) [172]	160	32	64 Bit	5 Bit	13 Bit	32 Bit

3-Address Instruction Set (Example)

Befehlssatz einer einfachen 3-Adressarchitektur

Befehl		Funktion		Befehlscode
sub	rD, $rSimm_1$, rS_2	Subtraktion: oder	$rD = rS_1 - rS_2$ $rD = \text{imm}_1 - rS_2$	000 | i | $rSimm_1$ | rS_2 | rD
ld	rD, [$rSimm_1$, rS_2]	Laden: oder	$rD = \text{mem}\,[rS_1 + rS_2]$ $rD = \text{mem}\,[imm_1 + rS_2]$	001 | i | $rSimm_1$ | rS_2 | rD
st	[$rSimm_1$, rS_2], rS_3	Speichern: oder	$\text{mem}\,[rS_1 + rS_2] = rS_3$ $\text{mem}\,[imm_1 + rS_2] = rS_3$	010 | i | $rSimm_1$ | rS_2 | rD
bmi	rS, *label*	Bedingter Sprung:	if ($rS < 0$) goto *label*+pc	0110 | rS | *label*
call	*label*	UP-Aufruf:	r0 = pc, goto *label*	1 | *label*
ret		Rücksprung: goto r0		0111 | 0000 | 0000 | 0000

Performance Criteria

Performance Criteria

- Clock frequency and clocks per instruction
  - clock frequency = 1/cycle time (τ)
  - Number of instructions per program (IC)
  - Average cycles per instruction (CPI)
- Execution time T
  - $T = \text{IC} \times \text{CPI} \times \tau$
  - $T = \text{IC} \times (p + m \times k) \times \tau$
    - p: number of processor cycles (decode, execute)
    - m: number of memory cycles
    - k: memory cycle time / processor cycle time

Dependency of the Performance Factors

System Attributes	Performance Factors				
		CPI			
	IC	p	m	k	τ
Instruction Set Architecture	×	×	~		~
Compiler	×	×	×		
Processor implementation and control		×			×
Cache and memory hierarchy				×	×

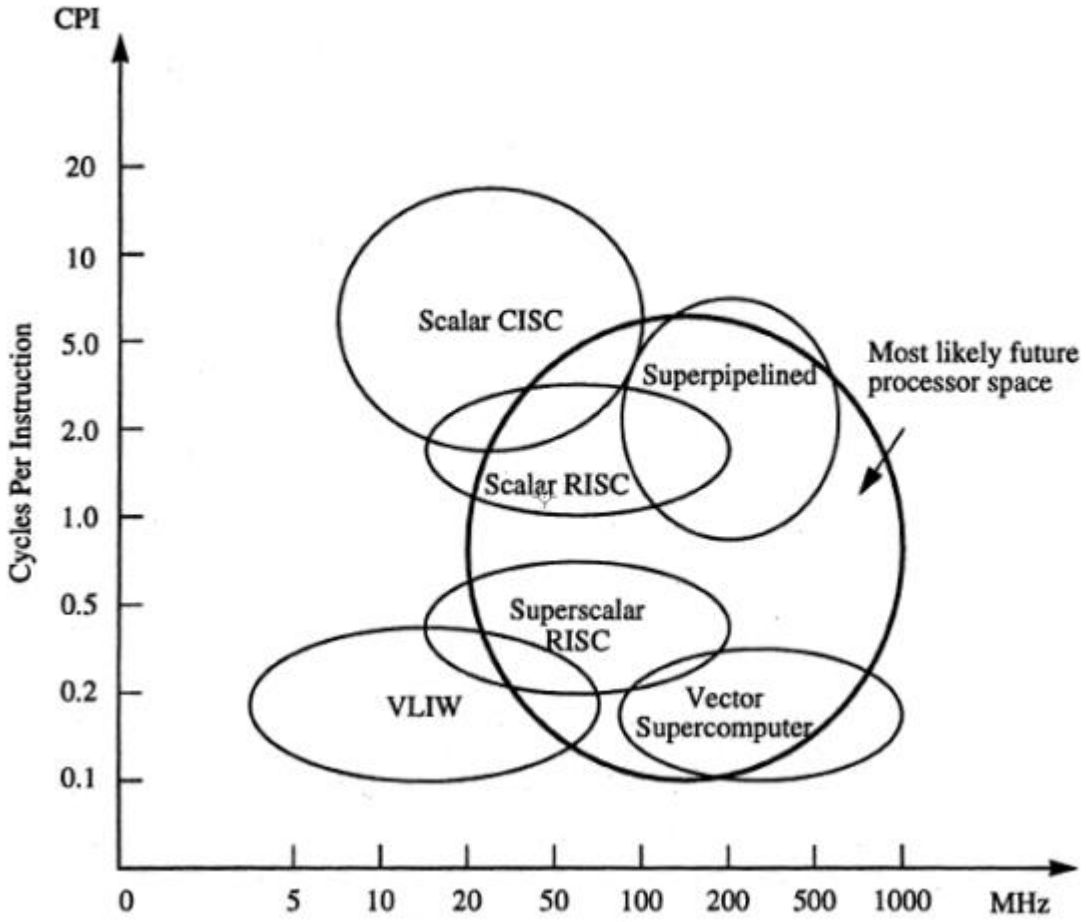
Data Path Implementation

Execution of Instructions

- *Fetch*: loading of next instruction
  - PC register (“program counter”) points to instruction, memory access required
- *Decode*: decoding the instruction
  - Determine also the operands
- *Execute*: execute the instruction (e.g. add, sub, branch, …)
- *Write-back*: store result
  - Store results in target register or memory

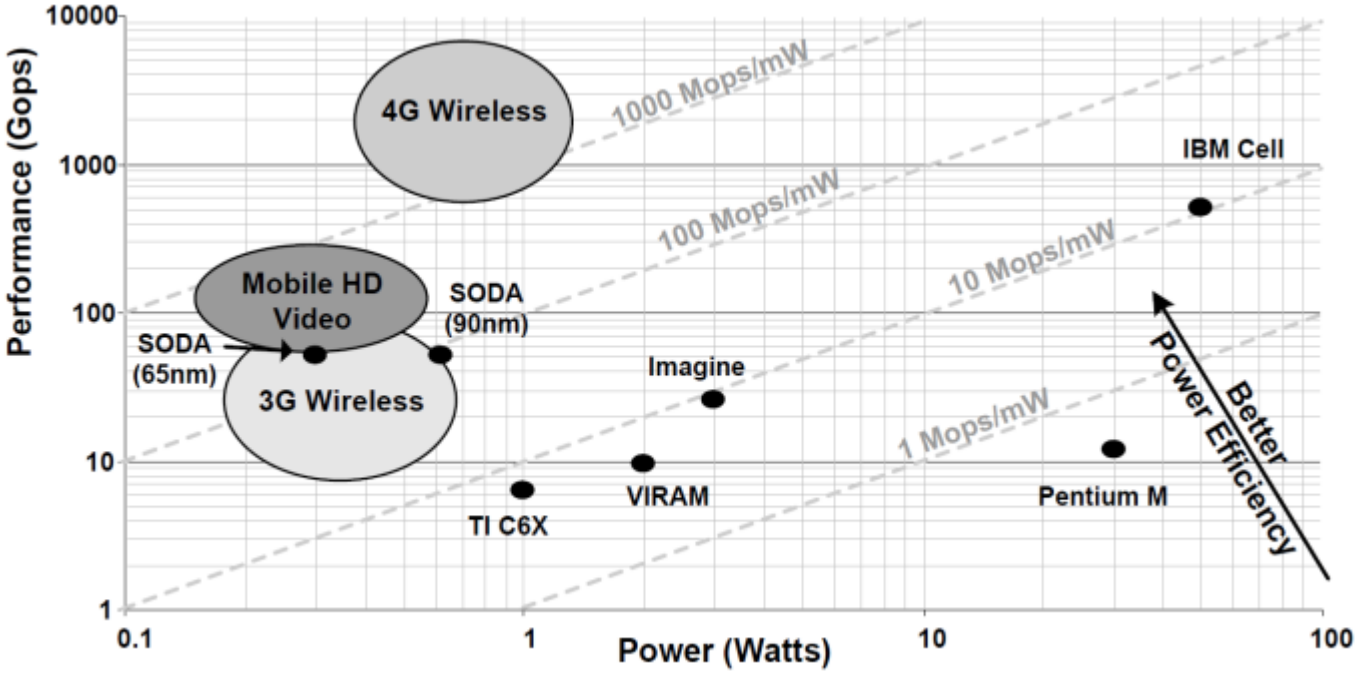
- *Several cycles are necessary (CPI > 1, typ. CPI ≈ 10)*

Design Space



Design Space

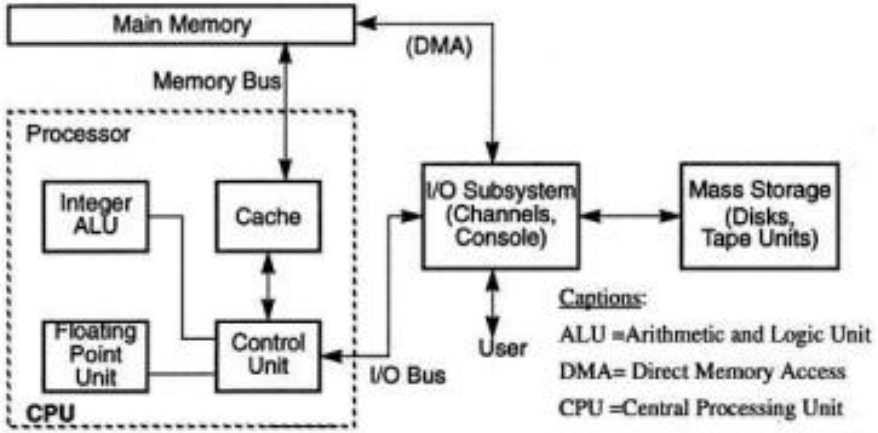
- Energy consumption per instruction



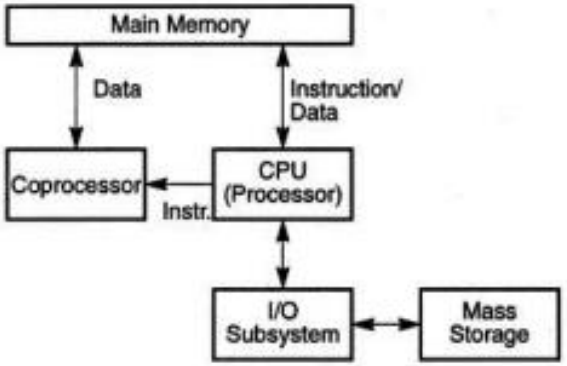
Scalar Processors

- “Scalar Processor” = 1 functional unit / CPU
  - e.g., integer unit or floating point unit
  - CPI ≥ 1
- Coprocessor
  - Floating-point, graphic, multimedia, …

Built-in FPU vs. Coprocessor



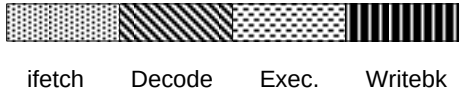
(a) CPU with built-in floating-point unit

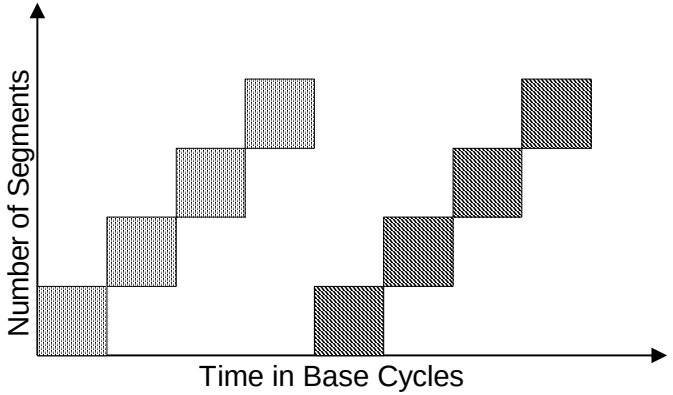


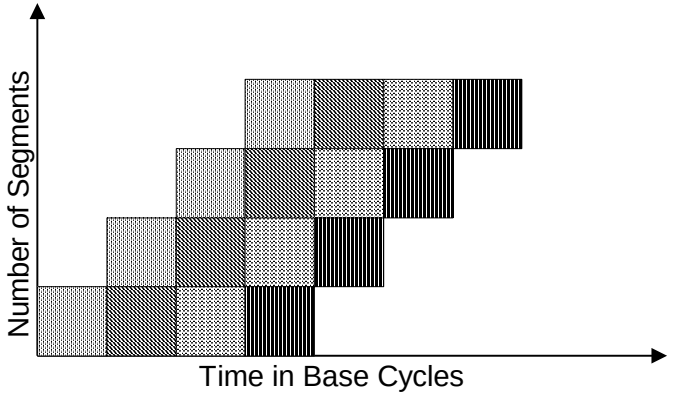
(b) CPU with an attached coprocessor

Basic instruction cycle and Pipelining

- Basic instruction cycle
  - fetch / decode / execute / write back
  - Some phases may take several base-cycles
- Pipelining
  - Overlapping of different phases of the execution
  - First result available after n base-cycles (n ... number of segments)
  - Subsequent results available after one additional cycle each
  - (Theoretical) maximum speedup is equal to the number of segments
  - Used with control-flow and arithmetic pipelines
  - Pipeline has to be flushed when asynchronous events take place





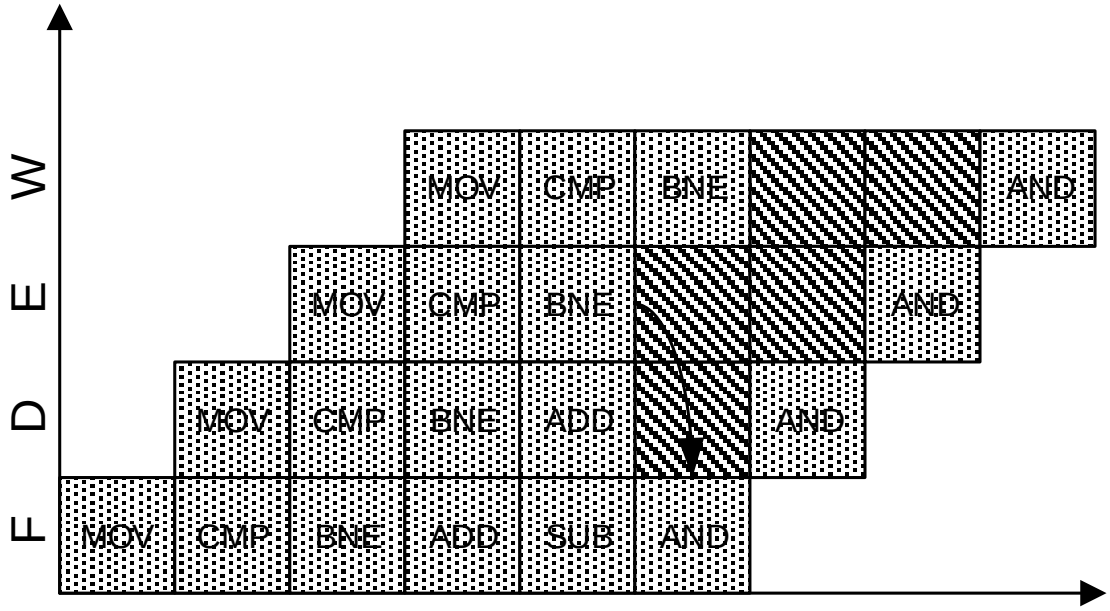


Pipelining issues

- Problem with control flow
  - Prerequisite: “next” instruction to execute is “next” instruction in memory
  - Branches (conditional and unconditional), calls, etc. violate this prerequisite
- => pipeline needs to be flushed and re-filled
- Objective: Avoid flushing the pipeline
  - Delayed branches
  - Branch prediction
  - Conditional execution

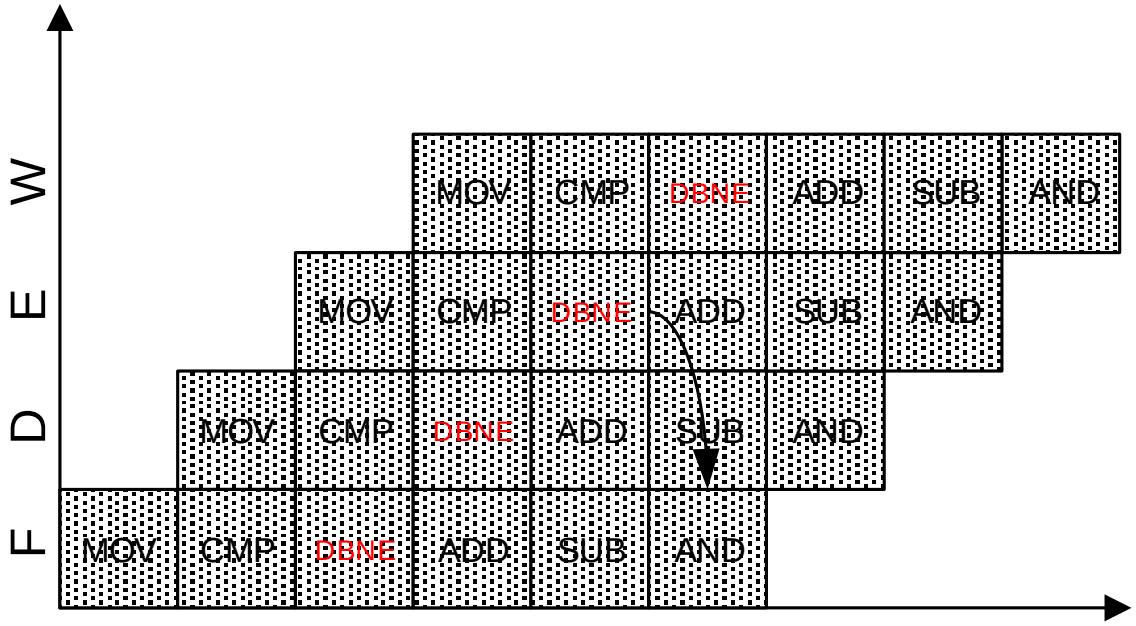
Branches & Instruction Pipeline

```
    MOV R8, [R0]
    CMP R1, R2
    BNE L1
    ADD R3, R4, R5
    SUB R4, R4, #1
    …
L1: AND R9, R10, R11
```



Delayed Branches

```
    MOV R8, [R0]
    CMP R1, R2
    DBNE L1
    ADD R3, R4, R5
    SUB R4, R4, #1
    …
L1: AND R9, R10, R11
```



- Branch happens “delayed”
- *Different OP-Code* for branch instruction required

Branch Prediction

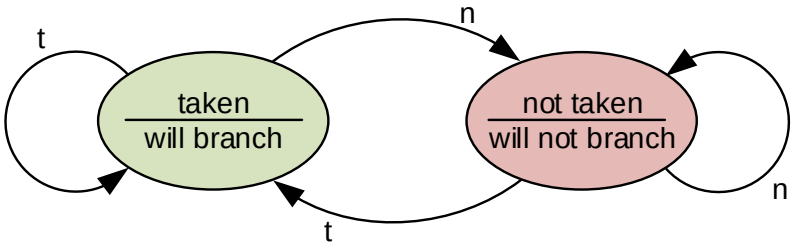
- Predict branch target
  - Observation: most branches are “easy to predict”

```
for (i = 0; i < 100; i++) {
    sum = x[i] * c[N - i];
}
```

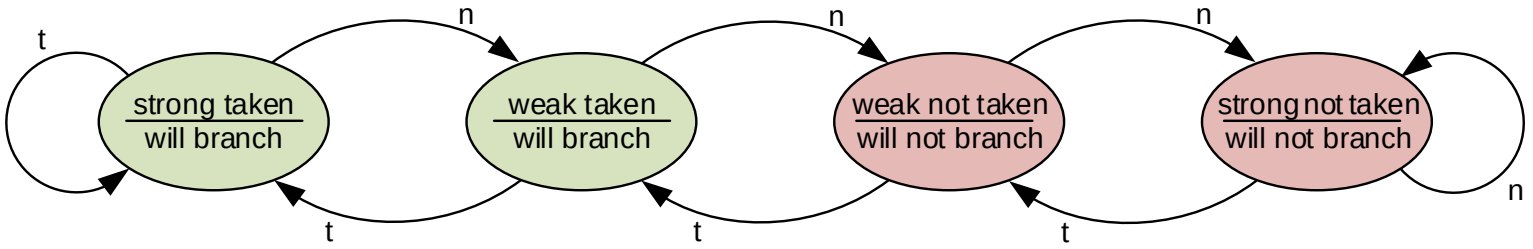
  - Only if prediction fails, pipeline must be flushed (predicted target ≠ “real” target)
- Prediction is based on historical data
  - Simple rule: “use same target as at last branch”
  - Exploit statistics on previous branching behavior, i.e., if a specific branch target has been chosen often it is likely that the same target will be chosen in the future

Branch Prediction Models

1-bit saturating counter (last outcome of the branch)

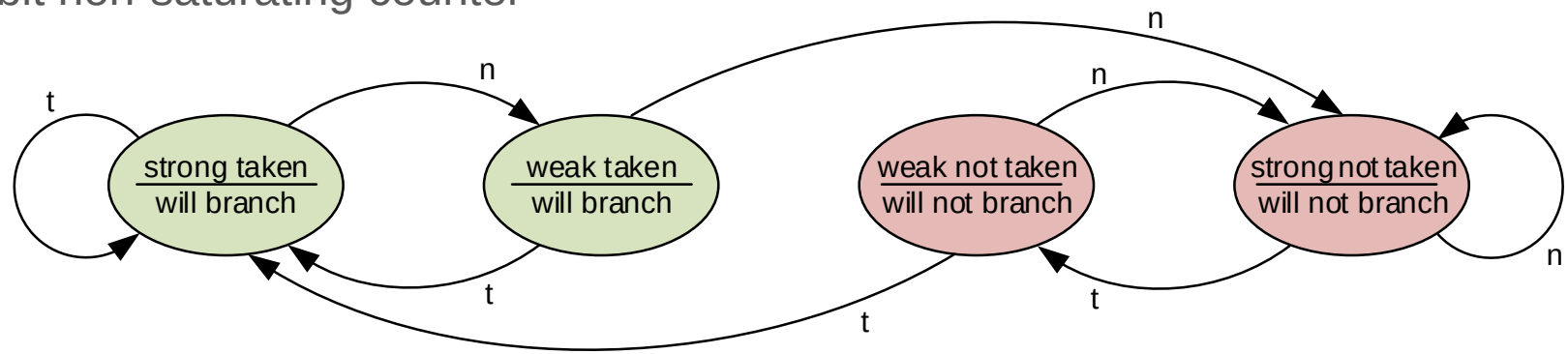


2-bit saturating counter, bimodal predictor – state machine

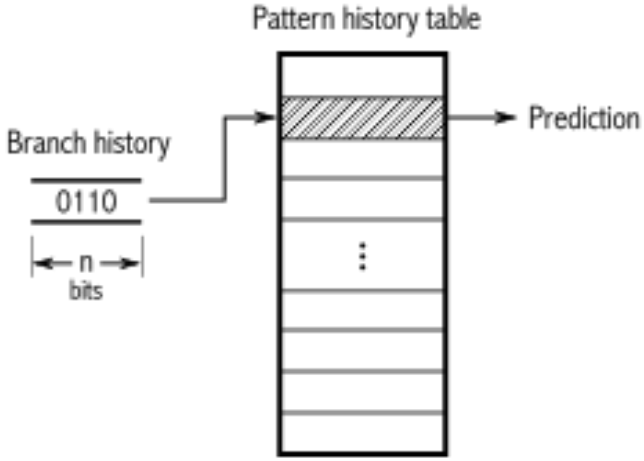


Branch Prediction Models

2-bit non-saturating counter



Two-level adaptive predictor



Conditional Execution

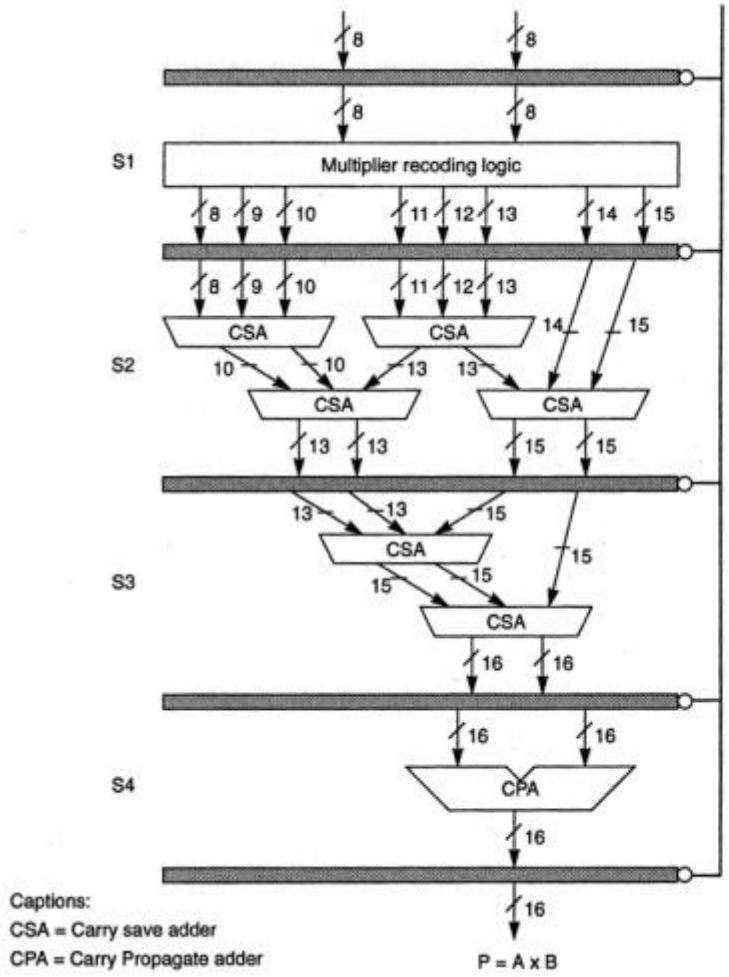
```
1 while (i != j) {
2     if (i > j) {
3         i -= j;
4     } else {
5         j -= i;
6     }
7  }
```

```
1 loop:   CMP  Ri, Rj          ; set condition "NE" if (i != j),
2                              ;                "GT" if (i > j),
3                              ;             or "LT" if (i < j)
4         SUBGT  Ri, Ri, Rj    ; if "GT" (Greater Than), i = i-j;
5         SUBLT  Rj, Rj, Ri    ; if "LT" (Less Than), j = j-i;
6         BNE  loop            ; if "NE" (Not Equal), then loop
```

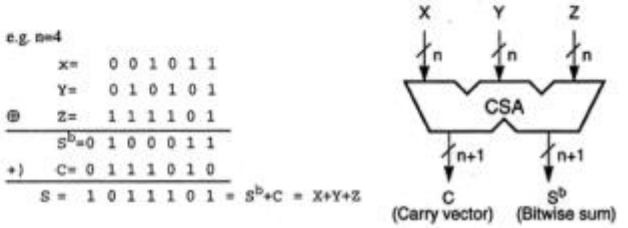
Other Pipelining Techniques

- Arithmetic Pipelining
  - E.g., for floating-point operations
- Super Pipelining
  - Additional partitioning of pipeline stages
  - Clocked at higher rates
- Linear/non-linear pipelines
- Address Pipelines
  - Determination of physical address

Arithmetic Pipelining



(a) An n-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique



(b) An n-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

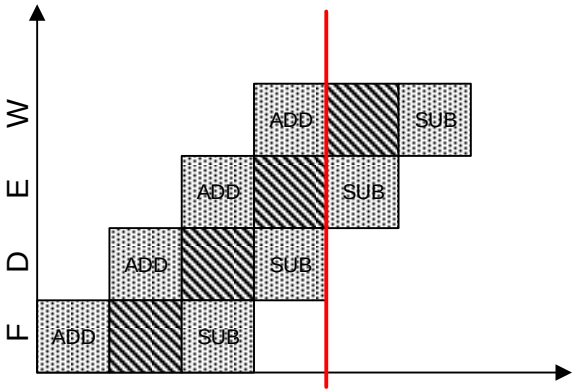
								1	0	1	1	0	1	0	1	=	A
							×)	1	0	0	1	0	0	1	1	=	B
								1	0	1	1	0	1	0	1	=	P_0
							1	0	1	1	0	1	0	1	0	=	P_1
						0	0	0	0	0	0	0	0	0	0	=	P_2
					0	0	0	0	0	0	0	0	0	0	0	=	P_3
				1	0	1	1	0	1	0	1	0	0	0	0	=	P_4
			0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_5
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_6
	+)	1	0	1	1	0	1	0	1	0	0	0	0	0	0	=	P_7
	0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1	= P

Issues with Pipelining

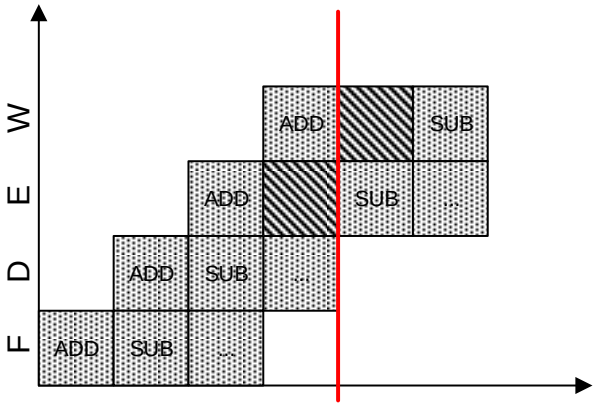
- Interlocking
  - either in Software
  - or in Hardware
- Bypass
  - Resolve data hazards
- Interrupt handling

Interlocking

```
add  r1, r2, r3
nop
sub  r3, r4, r1
...
```

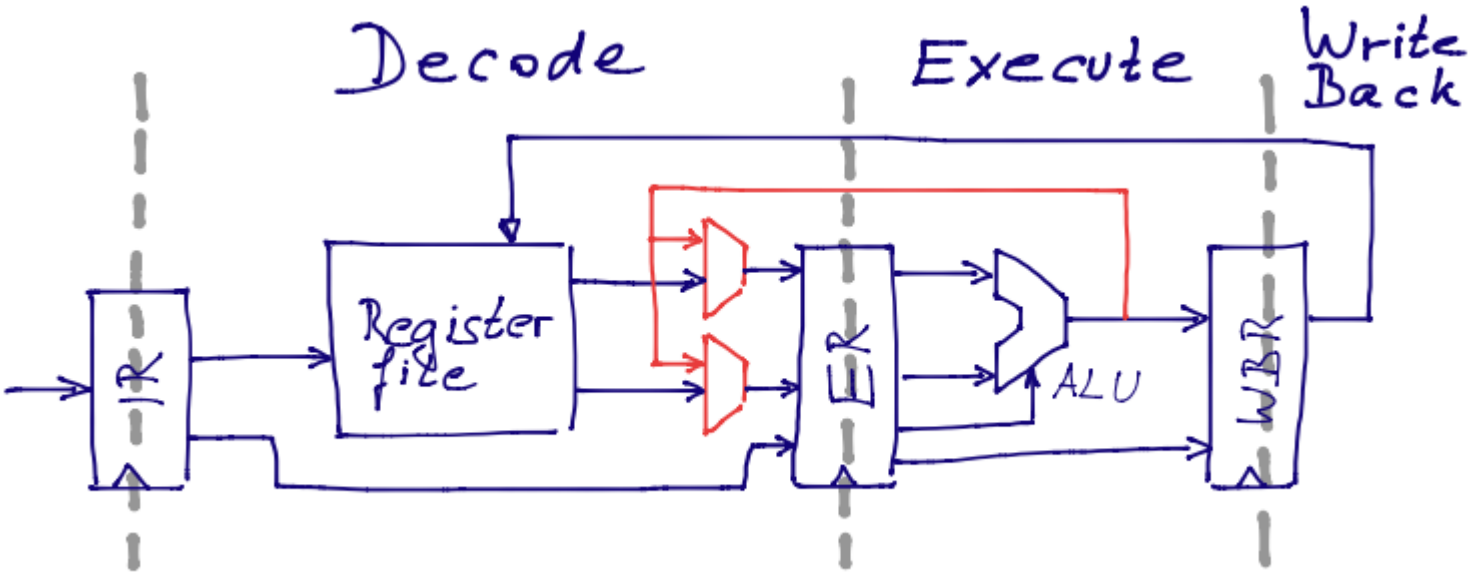



```
add  r1, r2, r3
nop
sub  r3, r4, r1
...
```



Bypass

```
add  r1, r2, r3
sub  r3, r4, r1
...
```



CISC vs. RISC

CISC vs. RISC

- Paradigm change for ISA (mid 1980s)
  - from *'Complex Instruction Set Architectures'* (CISC)
  - to *'Reduced Instruction Set Architectures'* (RISC)
  - RISC: all instructions have similar complexity
    => more feasible for pipelining
- Complexity of the ISA
  - Instruction format
  - Data format
  - Addressing modes
  - Registers

Features CISC

- Has moderate number of instructions
- Combines these instructions with numerous addressing modes (up to 13, can access memory, combines indexed, indirect; this is the main reason for complexity...)
- Sometimes orthogonal instruction-set (every instruction with every addressing mode)
- Calculation of the EA (= effective address) of operands is solved in microcode, which requires several steps / cycles if external access is necessary
- Medium number of registers (up to 16)
- Compact code (compression effect of addressing-modes)
- Sometimes variable opcode-size for compact code
- 8-32 bit register- / accu-size
- Can also have aggregate (=complex) instructions

Features RISC

- Higher number of (specialized) instructions
- Only some (typically 5) addressing modes, “reduced”
- “Load-Store-Architecture” – only few instructions can access memory
- No micro-code, therefore each cycle faster, but more code needed
- Evaluation of EA of operand requires several instructions
- Typically 32 registers (3 address-fields in opcode, 5 bit each)
- Typically 32 bit, but not exclusively
- Typically no variable code size
- Typically separate caches for instructions and data
- Typically larger (two times higher) code-size compared to CISC
- Specialized architectures also might have lower number of instructions

Comparison RISC - CISC

- Naming grown historically
- Reduced refers mainly to number of addressing modes
- Each cycle in RISC executes faster (10-20%)
- In average RISC requires double code-size
- RISC good with large caches and short programs (loops)
- Originally replacement of “microcode” against “cache”
- Small controllers still resemble original CISC
- CISC good with large opcodes and/or large data
- High-performance CISC:
  - Has separate caches
  - Has separate buses
  - Does pre-decode and pre-fetch of operands
  - Internally uses RISC-architecture

post-RISC Era

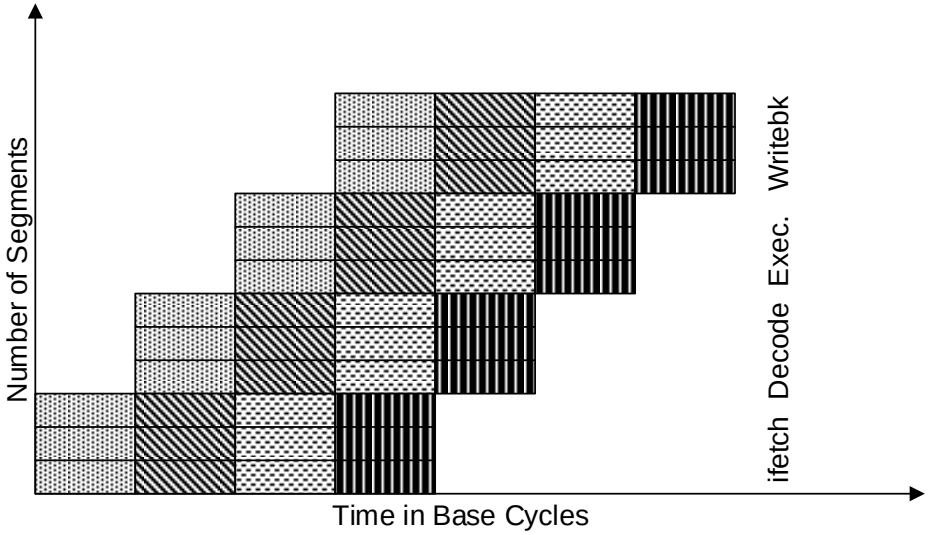
- Superscalar execution
- Out-of-order execution
- Branch prediction
- Additional instructions
- SIMD operations
- . . .

After all, whether RISC or CISC, modern processors use this techniques. Features are included in today's processors not because it's part of a design philosophy called "RISC", but because it enhances performance

Superscalar Processors

Superscalar Processors

- Multiple functional units
- Multiple (instruction-) Pipelines
  - Instructions can be executed on multiple functional units in parallel => CPI < 1 possible
  - On n functional units (n pipelines), up to n instructions per clock can be executed



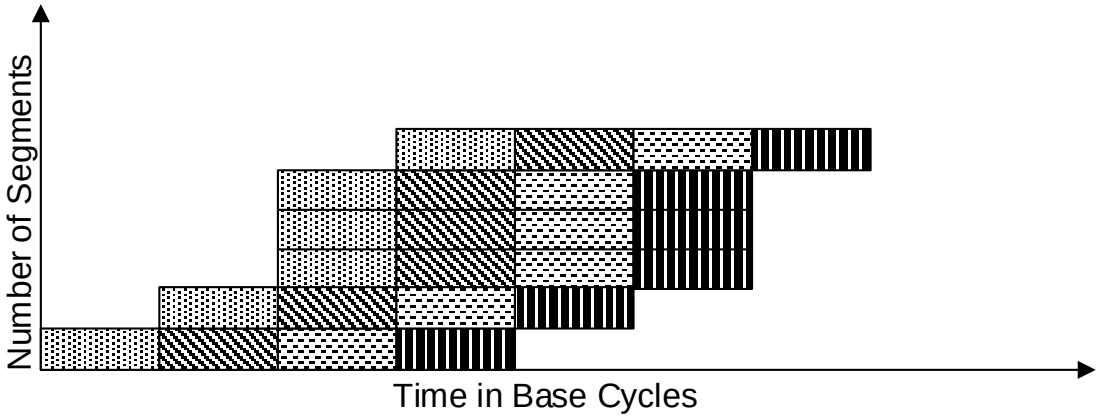
Superscalar Processors

- Processor “decides” which instructions can be executed in parallel
  - Implemented in hardware, dynamically scheduled
- Data dependencies among instructions must be considered!
- All current “desktop CPUs” are superscalar processors

Very Long Instruction Word Processors (VLIW)

VLIW Processors

- Multiple functional units
- Pipelining
- Common register file



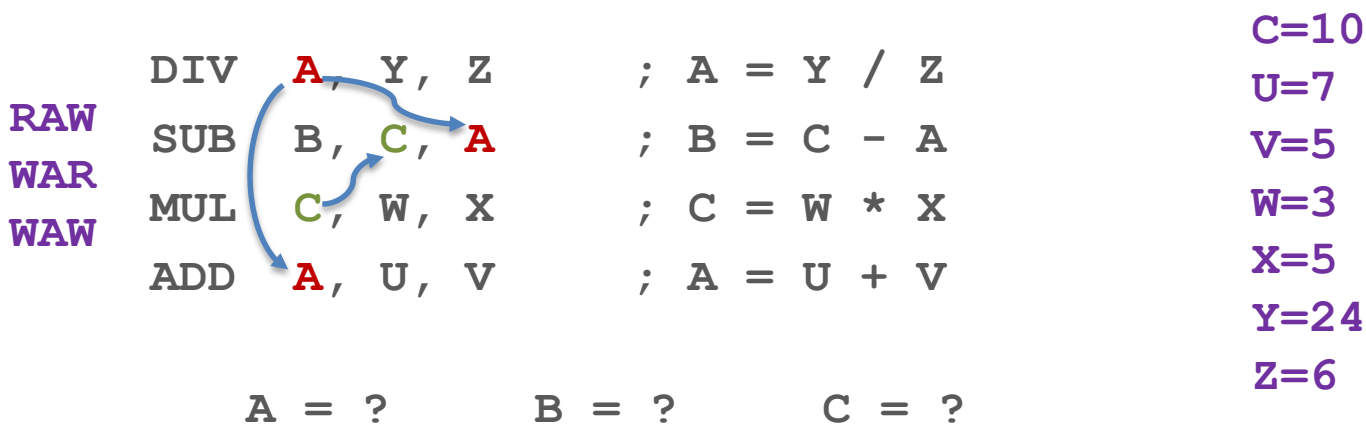
- Static assignment of instructions to functional units (at compile time)

VLIW Processors

- Very long instruction format (e.g., 1024 bit)
  - Increased program memory
  - Can be reduced for idle units
- Simpler hardware architecture (compared to superscalar proc.)
- Performance strongly dependent on compiler
- Explicit parallelism

Detour: Programming Model / Program Execution

A Few Assembler Lines



- Semantic of a program is defined by the *operations* and *data dependencies* between them.
  - Read-after-write (RAW)
  - Write-after-read (WAR)
  - Write-after-write (WAW)

Out-of-Order Execution

Out-of-Order Execution

- Properties
  - Until now: rigid sequential execution order of instructions
  - External memory access can cause long delays/stalls in the pipeline
- Principle
  - Instruction fetch/decode: in-order, potentially multiple instructions
  - Pool of instructions
  - Out-of-order execution (considering data dependencies!)
  - In-order write-back

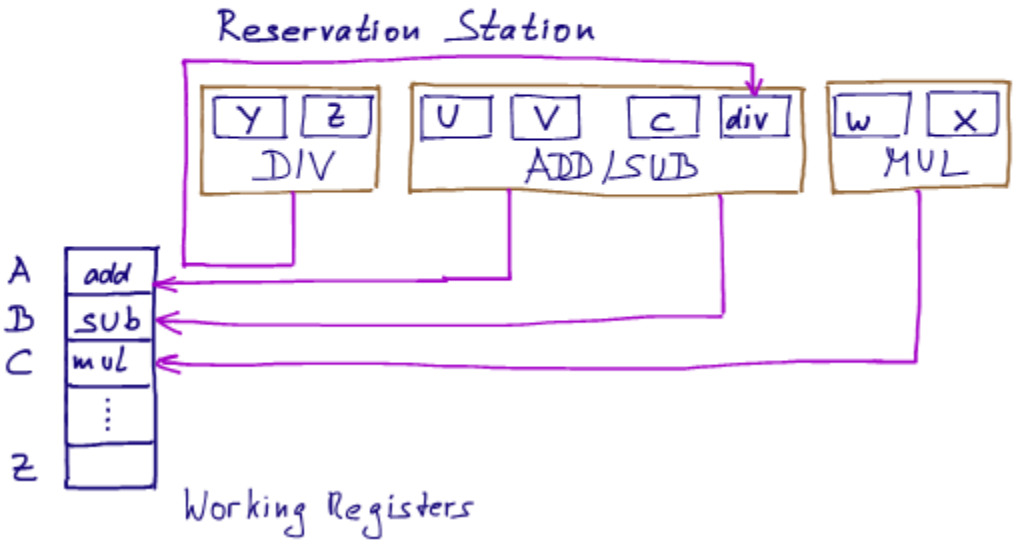
Out-of-Order Execution Techniques

- Parallel execution
  - Scoreboard
    - Additional bits for every register:
      indicate validity, read-tag, preference bit (r/w), . . .
  - Reservation station (Tomasulo Algorithm)
    - Working registers & reservation registers
- In-order write-back
  - Register renaming, re-order buffer
  - History buffer

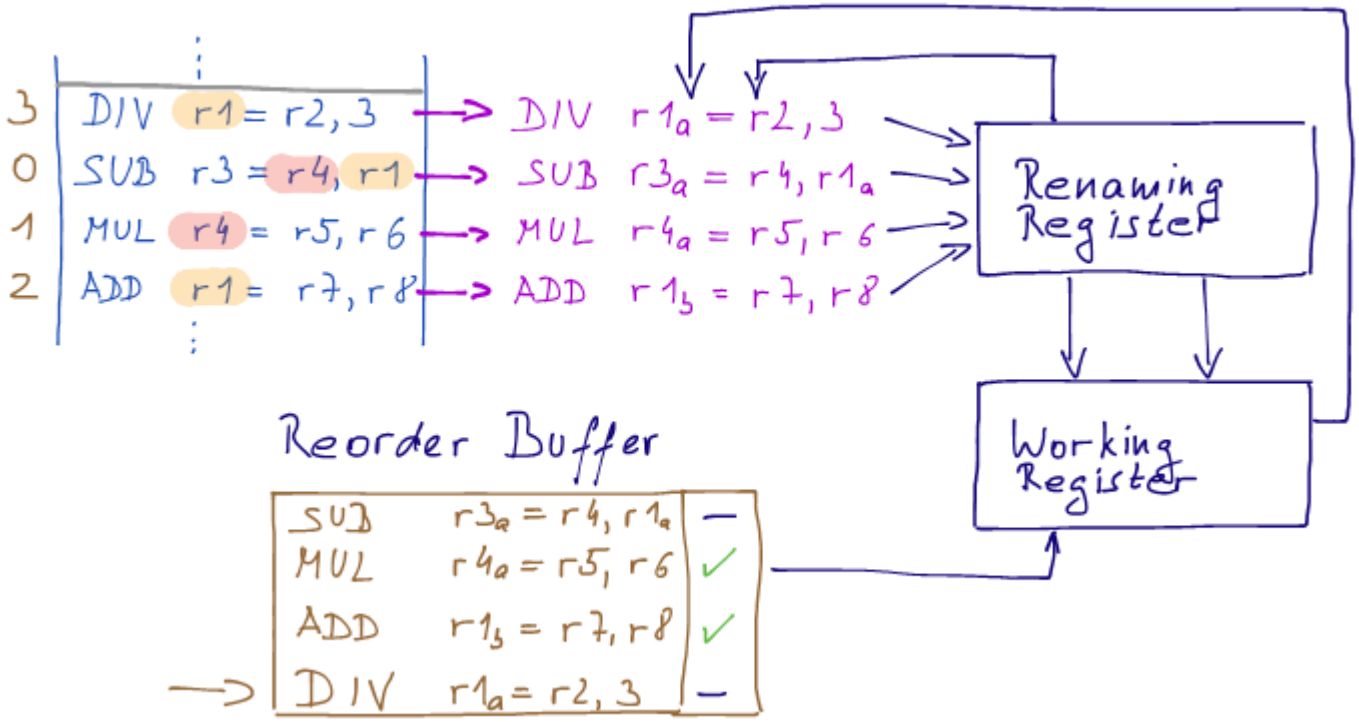
Tomasulo Algorithm

1: A = Y/Z
2: B = C − A
3: C = W · X
4: A = U + V

4 || 3 || 1 ; 2



Register Renaming & Re-order Buffer



Out-of-order Execution

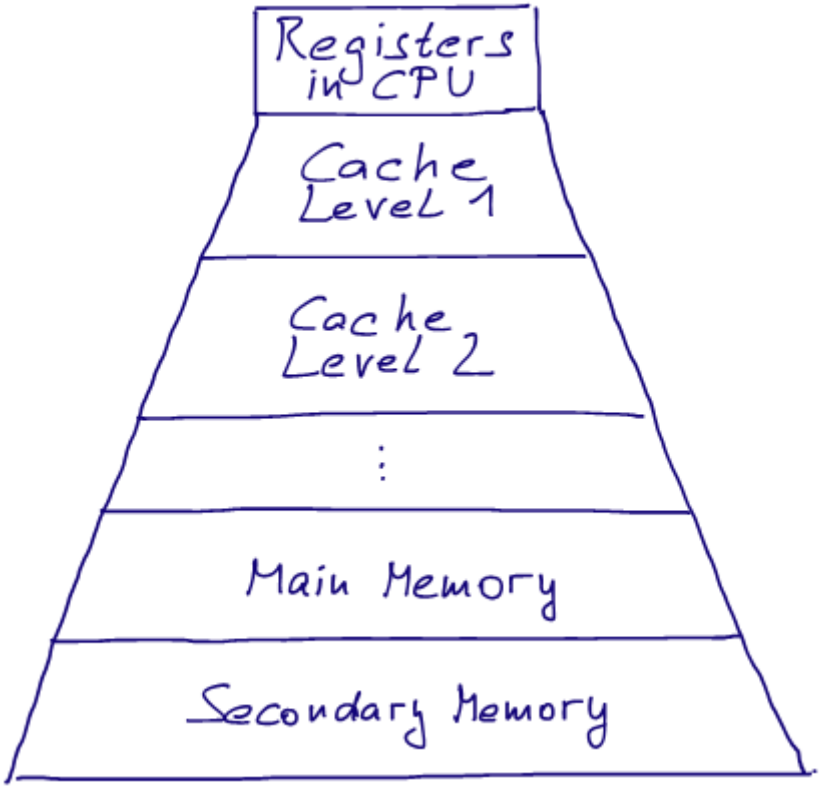
- Issues
  - Data dependencies
  - Different number of pipeline stages
  - Consistent “observable” processor state
  - Interrupt & Exception handling

Memory System

Memory System

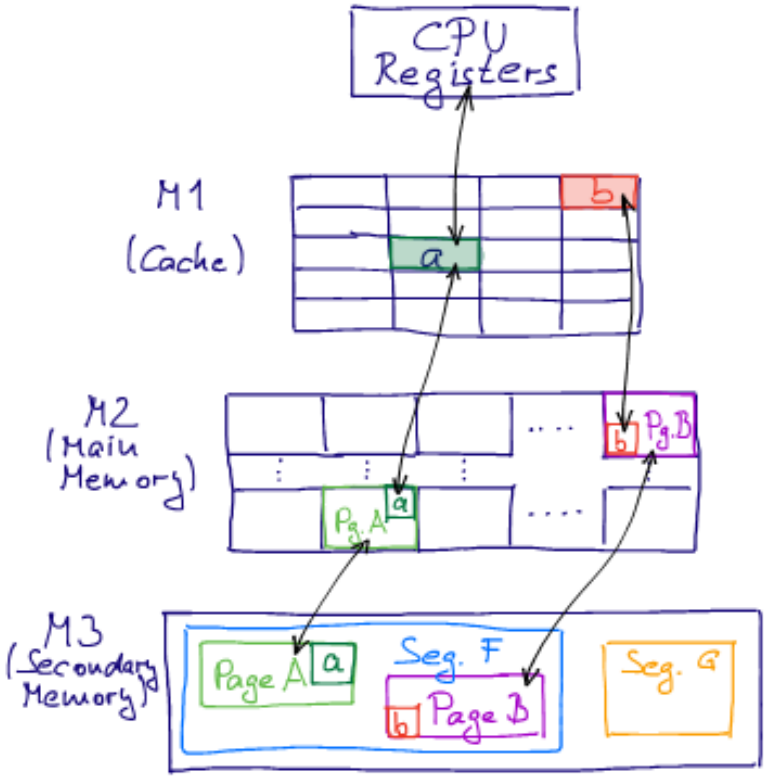
- Memory parameters
  - Access time
  - Storage size
  - Cost per byte
  - Transmission bandwidth
  - Transmission unit
  - Persistency
- Hierarchical organization

Memory Hierarchy



Memory Hierarchy: Inclusion

$M1 \in M2 \in \ldots \in Mn$



Cache

- Cache is a small *high-speed memory*
- Cache memory helps to reduce the time for accessing data CPU ⇔ Main memory
- Reduce access time based on *“locality of reference”*

Memory: Coherence

- Copies of same data must remain consistent on higher layers
  - Example: Modifications of cached data
    => update modified data on higher layers
- Cache strategies
  - Write-through (WT): immediate update
  - Write-back (WB): delayed update
- Cache coherency important in multi-core systems

Cache & Locality

- Programs (typ.) spend 90% of the time in 10% of the code
- *Temporal locality*
  - LRU algorithm, working sets, loops, stacks
- *Spatial locality*
  - Used memory regions are in close proximity
  - Arrays, structs, . . .
- *Sequential locality*
  - Instructions are executed sequentially (except branches)

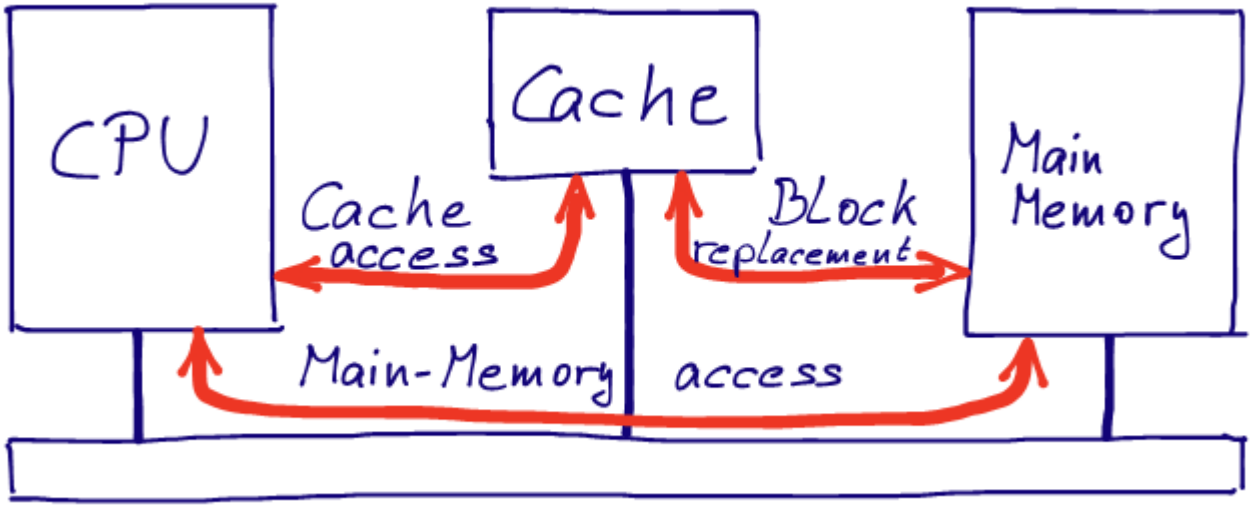
Cache Cycles

- CPU performs read or write
- Cache intercepts the bus transaction & decreases response time
- *Cache Hit:* When ever the cache contains the information requested
- *Cache Miss:* When ever the cache does not contain the information requested
- *Cache Consistency:* Cache always reflects what is in main memory
  - Snoop: Cache is watching the address lines for transactions
  - Snarf: When a cache takes the information from the data lines
  - Dirty data: When data is modified within cache but not modified in main memory
  - Stale data: When data is modified within main memory but not modified in cache

Cache Architecture

- Read architectures: “Look Aside” or “Look Through”
- Write policies: “Write-Back” or “Write-Through”
- *Look Aside:* sits in parallel with main memory
  - Both, main memory and cache see a bus cycle at the same time
  - Look aside caches are less complex and less expensive
  - The architecture provides better response to cache miss because both, DRAM and cache, see the bus cycle at the same time
  - *Main drawback:* processor can not access cache while other bus master is accessing main memory.

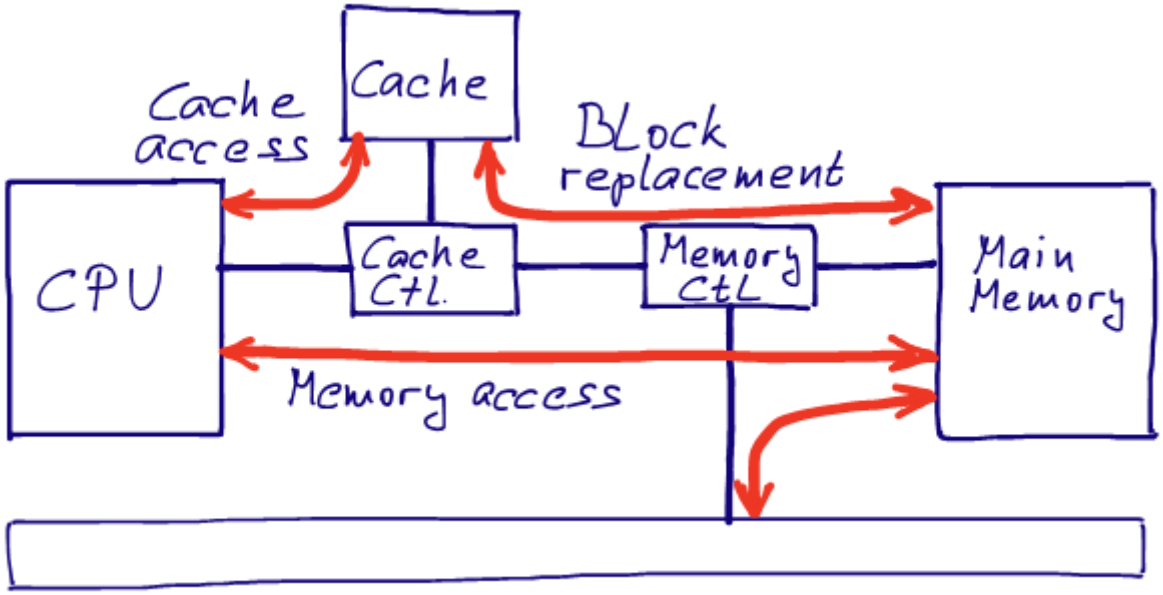
Cache: Look Aside Architecture



Cache Architecture

- *Look Through:* cache sits between processor and main memory
  - The cache sees the processor bus cycle before allowing it to pass on to the system bus
  - This architecture allows the processor to read out of cache while another bus master is accessing the main memory
  - This cache architecture is more complex and more costly
  - Another downside is that memory access on cache misses are slower because main memory is not accessed until after the cache is checked.

Cache: Look Through Architecture



Cache Architecture: Write Policy

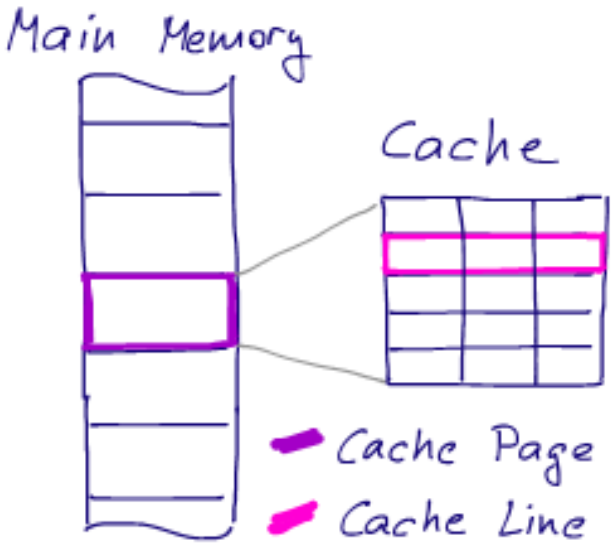
- *Write-back policy:* Cache acts like a buffer
  - Write cycle updates the data in the cache
  - Update of main memory is performed later (cp. “dirty data”)
  - Might reduce main memory updates, but increases complexity
- *Write-through policy:* The processor writes through the cache to main memory
  - Write cycle updates both cache and memory
  - Access to main memory at each write (no “dirty data”)
  - Simpler but less effective

Cache Components

- *SRAM:* Static Random Access Memory (SRAM) is the memory block that holds the data (== cache size)
- *Tag RAM:* stores the address of the data that is currently stored in the SRAM
- *Cache controller:* implements the cache policies
  - Updates SRAM and TRAM
  - Read/write policy

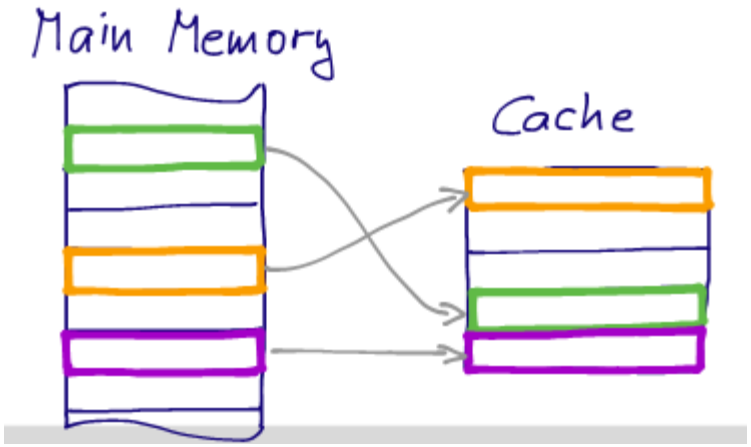
Cache Organization

- Main memory is divided into equal pieces (Cache Page)
  - Size of a page is dependent on the size of the cache
- Cache page is broken into smaller pieces (Cache Line)
  - Cache line is determined by processor and cache design



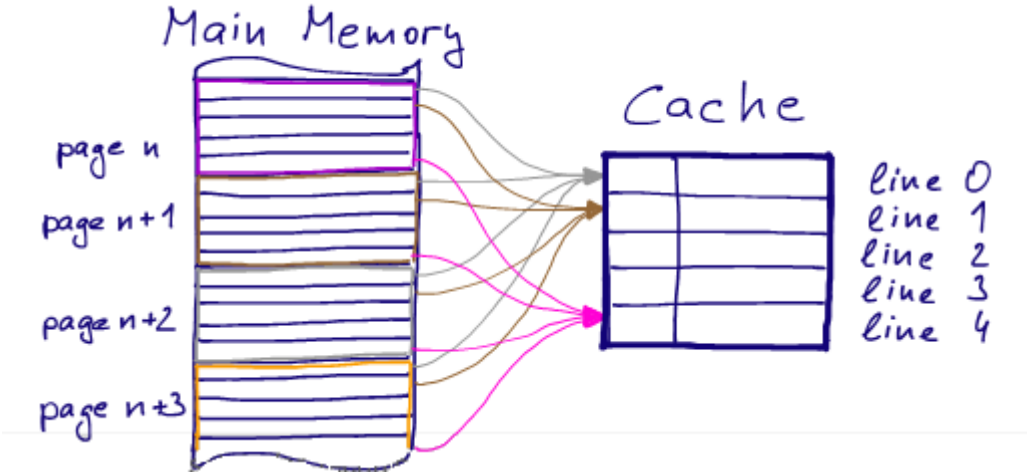
Fully-Associative Cache

- Any line in main memory is allowed to store in any location in the cache
  - Disadvantage: complexity of implementation
  - TRAM access time is critical for overall performance



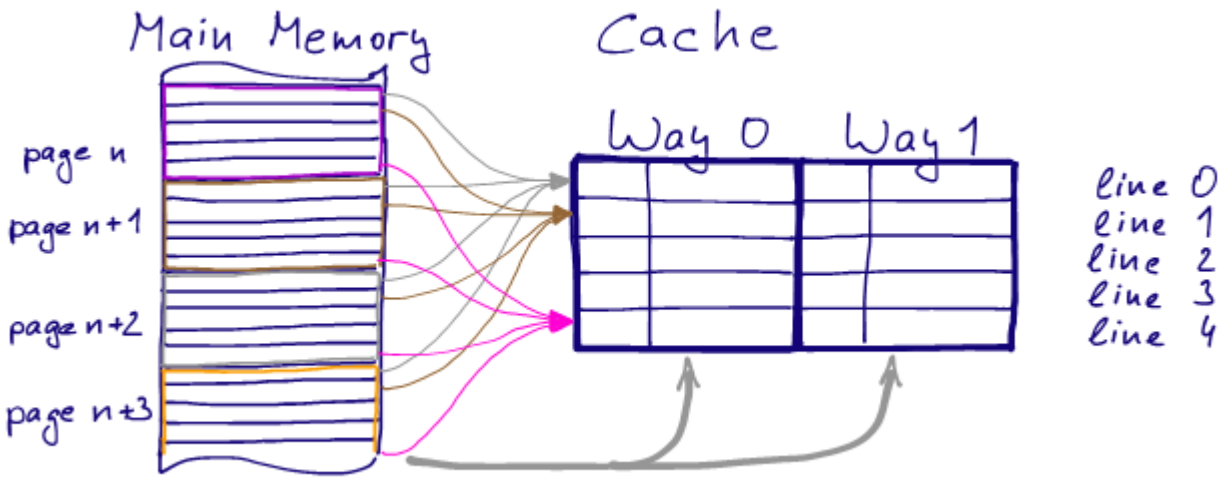
Direct-Mapped Cache

- Main memory is divided into cache pages. Size of each page is equal to the size of the cache
  - Unique cache line for line i of all pages
  - It is the least complex of all three caching schemes



Set-Associative Cache

- Cache SRAM is divided into equal sections called *cache ways*
  - Cache page is equal to the size of the cache way
  - Each cache way is treated like a small direct mapped cache
  - n lines of memory may be stored at any time
  - Helps to reduce trashing (loading/replacing the same line)



Direct Memory Access

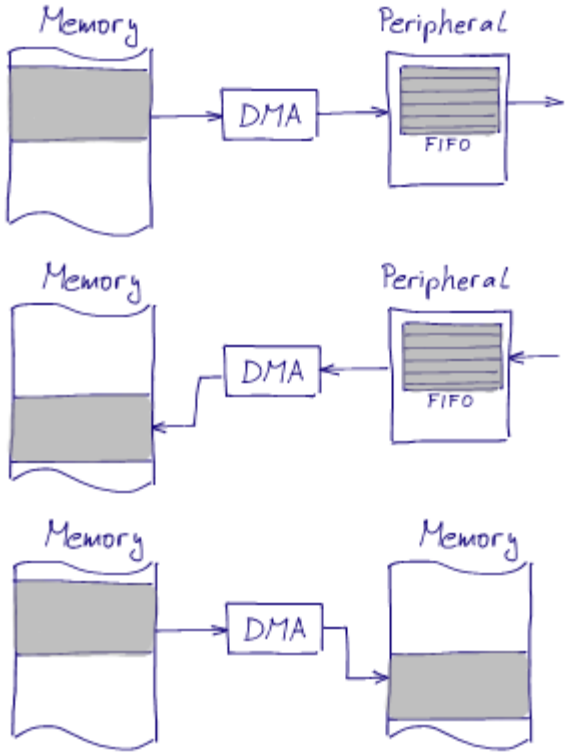
Data Transfer Mechanisms

- Polling
  - Processor is dedicated to acquire data
  - Not efficient, processor cannot do other tasks
- Interrupts
  - Processor is interrupted when incoming data is ready & acquires data
- Direct Memory Access
  - Dedicated unit for data acquisition
  - Reads incoming data and stores the data in system memory
  - Processor can work on other tasks in parallel

Direct Memory Access

- External peripherals communicate with DMA controller for data transfer
- DMA controller manages several channels for data transfer and DMA requests
- Channels must be enabled by the processor for DMA controller to respond to DMA requests

DMA Data Flow



DMA Transfer Types

- Fly-by DMA Transfer
  - Fastest data transfer, single cycle or single address transfer
  - DMA controller supplies the address, device reads or writes the data
  - Memory-to-memory transfers are not possible!
- Fetch-and-deposit DMA transfer
  - *Dual-cycle* or dual address transfer
  - 1st cycle: Data is read into a *temporary register* of DMA controller
  - 2nd cycle: Data is written to memory or other device
  - Used for different bus widths

DMA Applications

- Network cards
- Intra-chip data transfer (multi-core processors)
- Graphic cards
- Disk drive controllers
- Sound cards
- . . .

Conclusion