

# Embedded System Hardware - Processing (Part II)–

Jian-Jia Chen  
(Slides are based on  
Peter Marwedel)  
Informatik 12  
TU Dortmund  
Germany

2018年 11 月 14 日

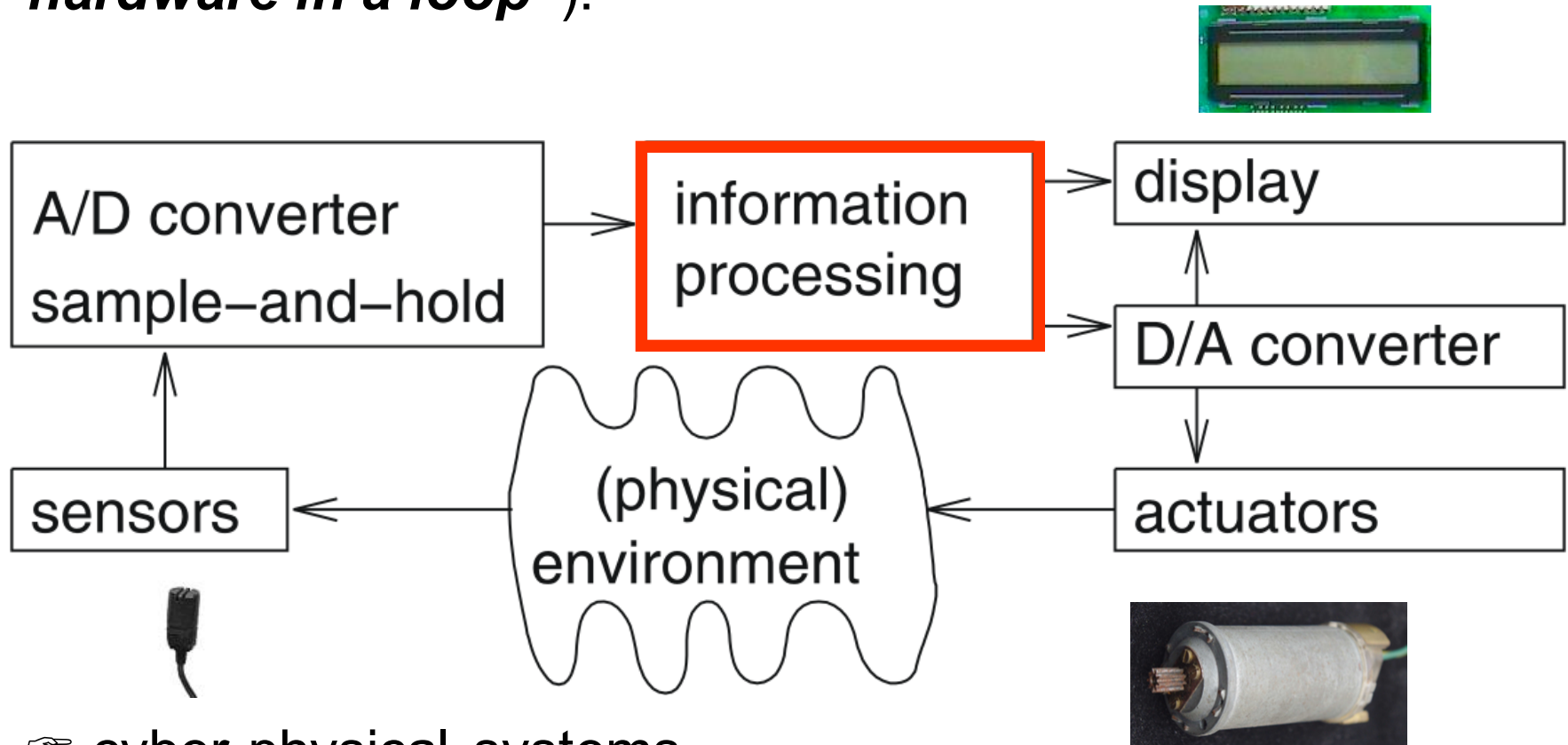


© Springer, 2018

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.

# Embedded System Hardware

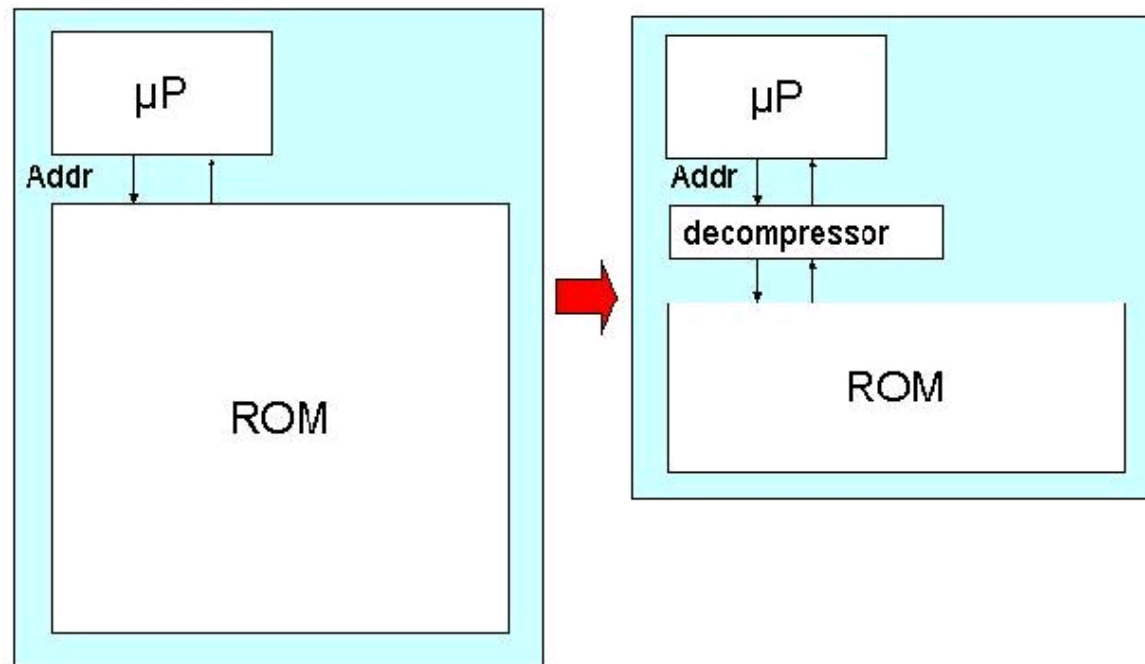
Embedded system hardware is frequently used in a loop (***“hardware in a loop”***):



👉 cyber-physical systems

# Key requirement #1: Code-size efficiency

- Overview: <http://www-perso.iro.umontreal.ca/~latendre/codeCompression/codeCompression/node1.html>
- **Compression techniques: key idea**

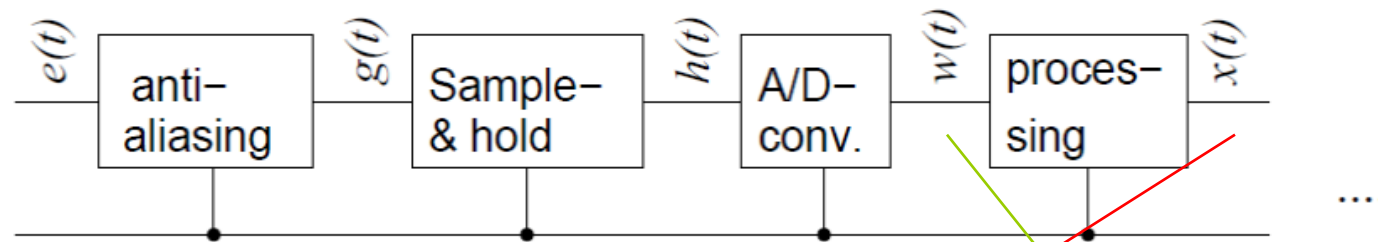


## Key requirement #2: Run-time efficiency

- Domain-oriented architectures -

---

Example: Filtering in Digital signal processing (DSP)

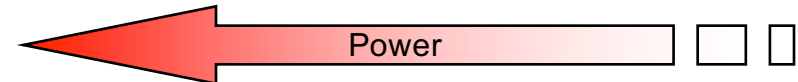


$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

Signal at  $t=t_s$  (sampling points)

# Key requirement #3: energy-efficient and power efficient

Execution platform	Relevant during use?		
	Plugged	Uncharged periods	Unplug- ged
E.g.	Factory	Car	Sensor
Global warming	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Cost of energy	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Increasing performance	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Problems with cooling, avoiding hot spots	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Avoiding high currents & metal migration	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Reliability	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Energy a very scarce resource	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>



# Key Requirement #4: Real-time capability

---

- **Timing behavior has to be predictable**

Features that cause problems:

- Unpredictable access to shared resources
  - Caches with difficult to predict replacement strategies
  - Unified caches (conflicts between instructions and data)
- Pipelines with difficult to predict stall cycles ("bubbles")
- Unpredictable communication times for multiprocessors
- Branch prediction, speculative execution
- Interrupts that are possible any time
- Memory refreshes that are possible any time
- Instructions that have data-dependent execution times

👉 **Trying to avoid as many of these as possible.**

# DSP and Multimedia Processors



© Springer, 2010

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.

# Saturating arithmetic

- Returns largest/smallest number in case of over/underflows
- Example:

a		0111
b	+	1001
standard wrap around arithmetic		(1)0000
saturating arithmetic		1111
<b>(a+b)/2:</b> correct		1000
wrap around arithmetic		0000
saturating arithmetic + shifted		0111 “almost correct”

- Appropriate for DSP/multimedia applications:
  - No timeliness of results if interrupts are generated for overflows
  - Precise values less important
  - Wrap around arithmetic would be worse.

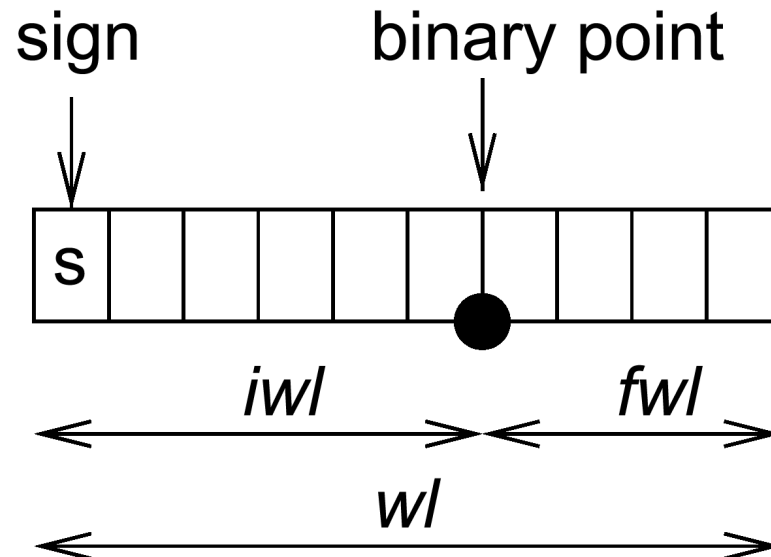


# On-Site Exercise

Suppose that the numbers in the following table are given. Please compute the result of the indicated operation using wrap-around and saturating arithmetic!

a		b		op	a op b	
(un)signed	bitvector	(un)signed	bitvector		wrap-around	saturating
signed	00110 <sub>2</sub>	signed	00110 <sub>2</sub>	+		
signed	01110 <sub>2</sub>	signed	01110 <sub>2</sub>	+		
signed	11110 <sub>2</sub>	signed	11110 <sub>2</sub>	+		
signed	01010 <sub>2</sub>	signed	01010 <sub>2</sub>	*		
unsigned	10110 <sub>2</sub>	unsigned	10110 <sub>2</sub>	+		

# Fixed-point arithmetic



Shifting required after multiplications and divisions in order to maintain binary point.

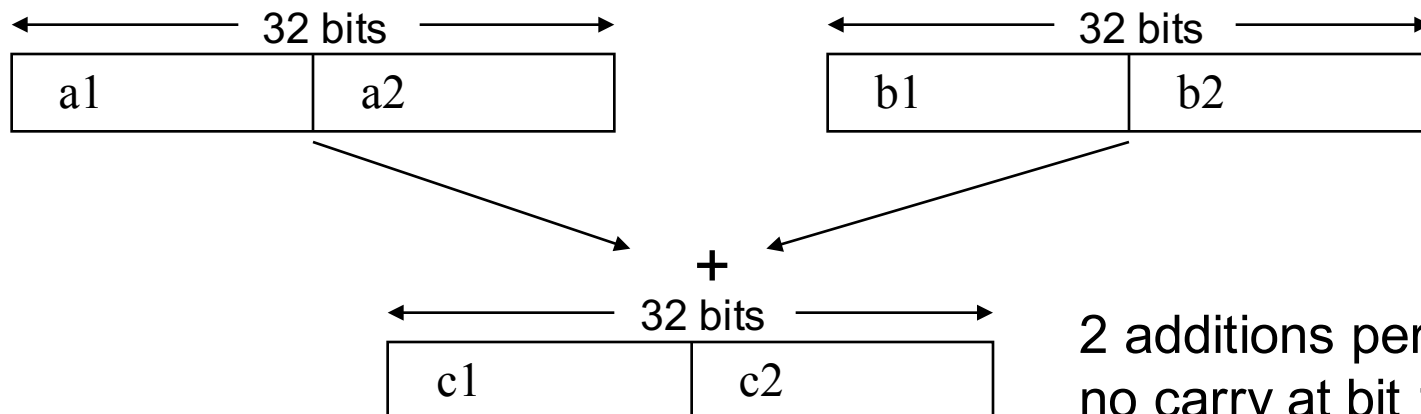
# On-Site Exercise

Suppose that the fixed-point numbers in the following table are given. Numbers are assumed to be signed. Please compute the result of the indicated operation!

a		b		op	a op b	
$(iwl, fwl)$	bitvector	$(iwl, fwl)$	bitvector		$(iwl, fwl)$	bitvector
(3,2)	00110 <sub>2</sub>	(3,2)	00110 <sub>2</sub>	+	(3,2)	
(3,2)	00110 <sub>2</sub>	(3,2)	00110 <sub>2</sub>	+	(4,1)	
(3,2)	00110 <sub>2</sub>	(4,1)	00110 <sub>2</sub>	+	(4,1)	
(3,2)	00010 <sub>2</sub>	(4,1)	00010 <sub>2</sub>	+	(2,1)	
(3,2)	00110 <sub>2</sub>	(4,1)	00110 <sub>2</sub>	*	(8,2)	

# Multimedia-Instructions, Short vector extensions, Streaming extensions, SIMD instructions

- Multimedia instructions exploit that many registers, adders etc are quite wide (32/64 bit), whereas most multimedia data types are narrow
- 👉 2-8 values can be stored per register and added. E.g.:



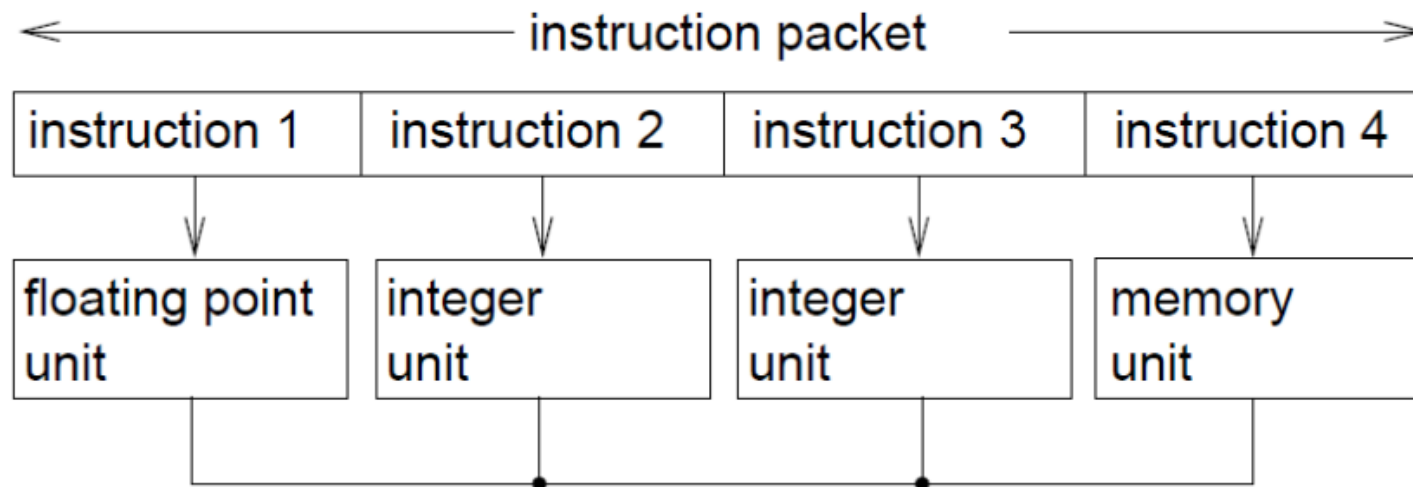
2 additions per instruction;  
no carry at bit 16

- Cheap way of using parallelism
- 👉 SSE instruction set extensions, SIMD instructions

# Key idea of very long instruction word (VLIW) computers (1)

---

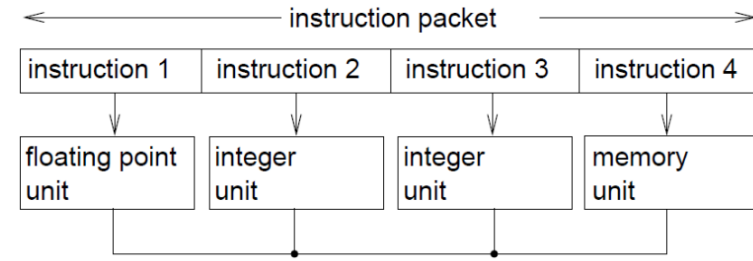
- Instructions included in long instruction packets.
- Multiple instructions are assumed to be executed in parallel.
- Fixed association of packet bits with functional units.



- Compiler is assumed to generate these “parallel” packets

# Key idea of very long instruction word (VLIW) computers (2)

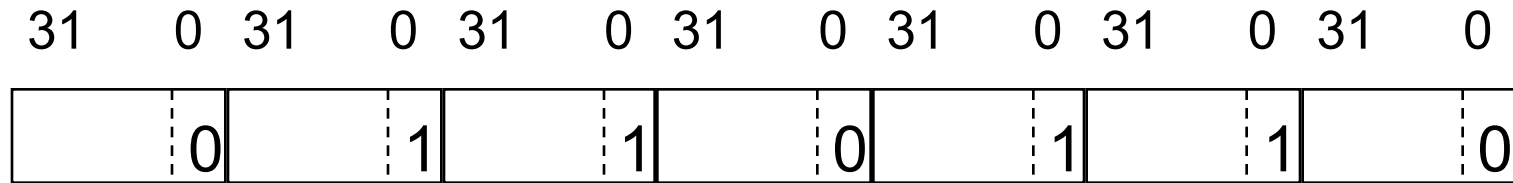
- Complexity of finding parallelism is moved from the hardware (RISC/CISC processors) to the compiler;



- Ideally, this avoids the overhead (silicon, **energy**, ..) of identifying parallelism at run-time.
- 👉 A lot of expectations into VLIW machines
- However, possibly low code efficiency, due to many NOPs
- 👉 Explicitly parallel instruction set computers (EPICs) are an extension of VLIW architectures: parallelism detected by compiler, but no need to encode parallelism in 1 word.

# EPIC: TMS 320C6xxx as an example

1 Bit per instruction encodes end of parallel exec.



Instr. A   Instr. B   Instr. C   Instr. D   Instr. E   Instr. F   Instr. G

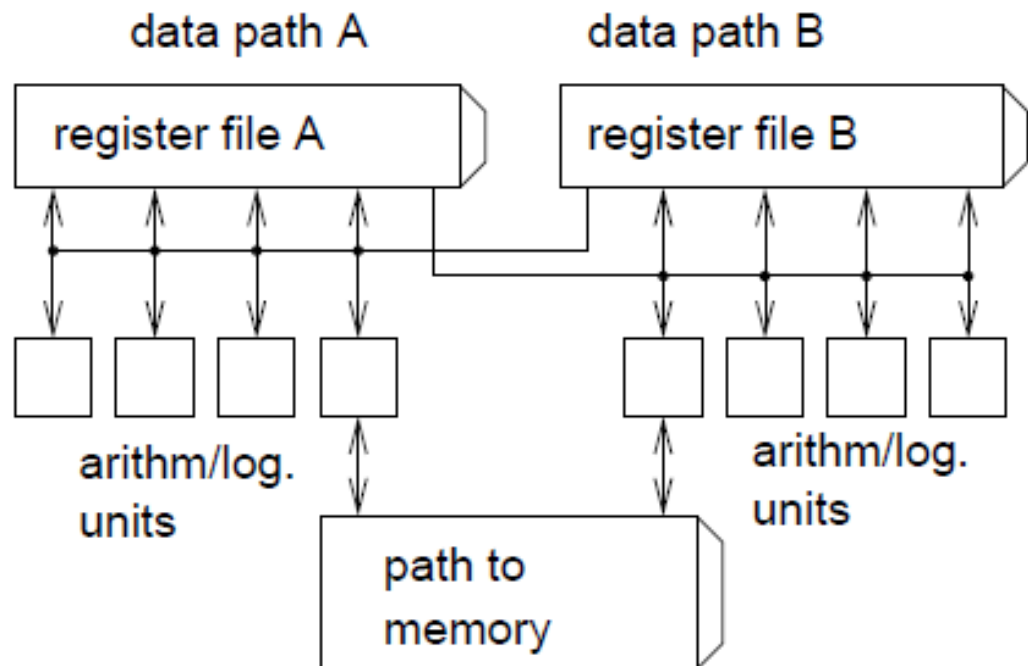
Cycle	Instruction		
1	A		
2	B	C	D
3	E	F	G

Instructions B, C and D use disjoint functional units, cross paths and other data path resources. The same is also true for E, F and G.

# Partitioned register files

- Many memory ports are required to supply enough operands per cycle.
- Memories with many ports are expensive.

☞ Registers are partitioned into (typically 2) sets, e.g. for TI C6xxx:





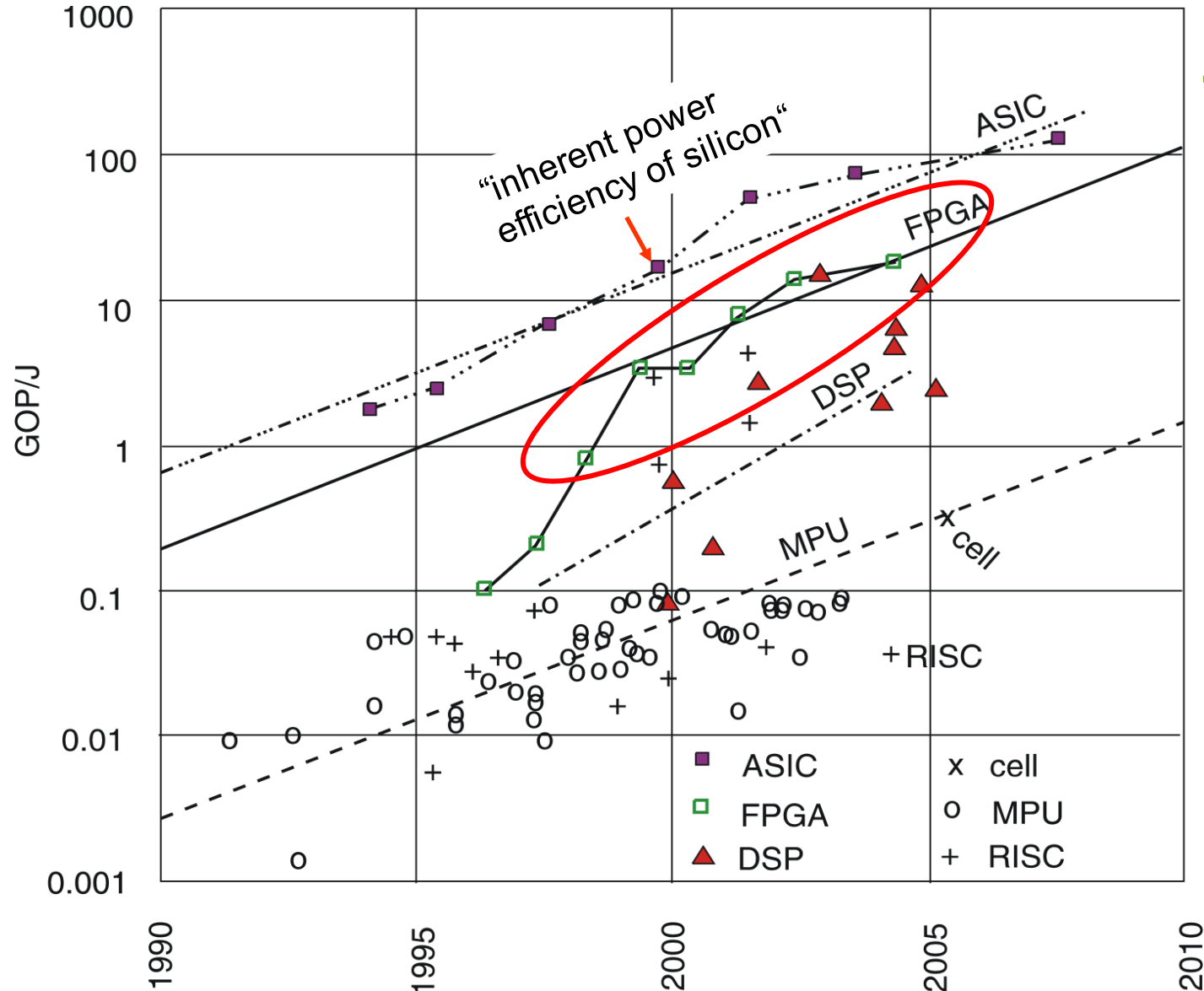
# Microcontrollers

## - MHS 80C51 as an example -

---

- 8-bit CPU optimised for control applications
- Extensive Boolean processing capabilities
- 64 k Program Memory address space
- 64 k Data Memory address space
- 4 k bytes of on chip Program Memory
- 128 bytes of on chip data RAM
- 32 bi-directional and individually addressable I/O lines
- Two 16-bit timers/counters
- Full duplex UART
- 6 sources/5-vector interrupt structure with 2 priority levels
- On chip clock oscillators
- Very popular CPU with many different variations

# Energy Efficiency of FPGAs



© Hugo De Man,  
IMEC, Philips, 2007

# Reconfigurable Logic

---

Custom HW may be too expensive, SW too slow.

Combine the speed of HW with the flexibility of SW

👉 HW with programmable functions and interconnect.

👉 Use of configurable hardware;

common form: field programmable gate arrays (FPGAs)

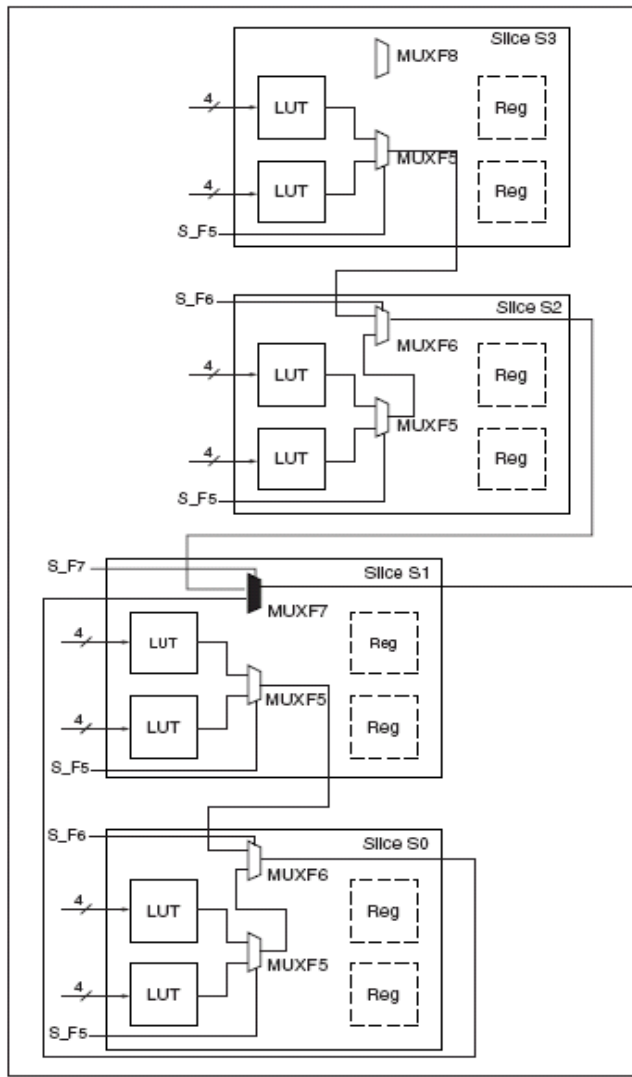
Applications:

- algorithms like de/encryption,
- pattern matching in bioinformatics,
- high speed event filtering (high energy physics),
- high speed special purpose hardware.

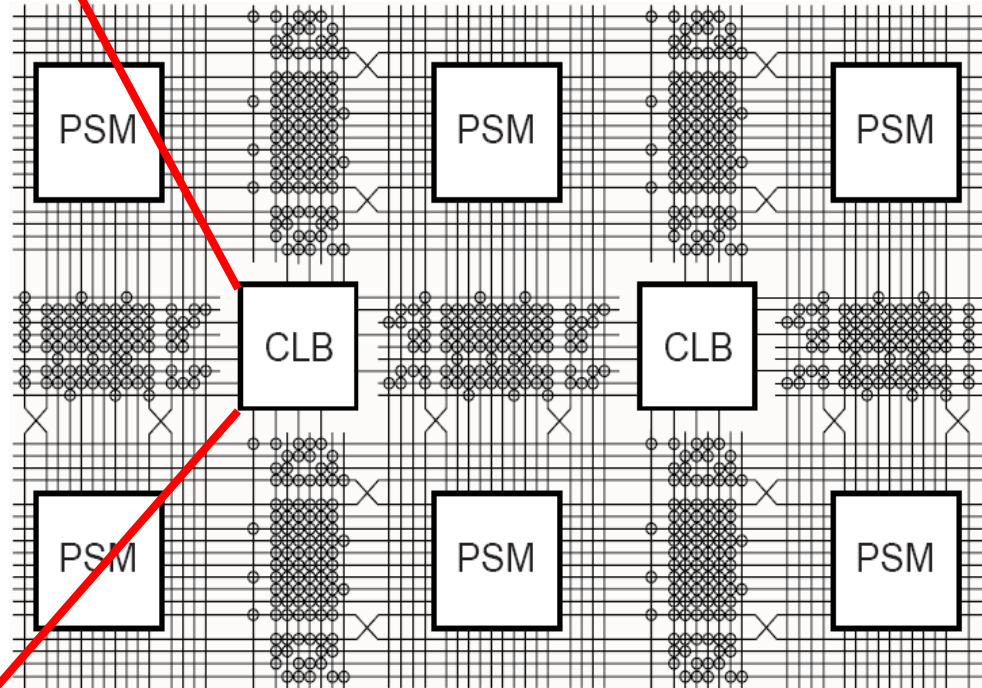
Very popular devices from

- XILINX, Actel, Altera and others

# Fine-Grained Reconfigurable Fabric



**CLB:** Configurable Logic Block  
**PSM:** Programmable Switch Matrix  
**Additionally:** I/O Blocks, RAM Blocks,  
Multiplier, CPUs, ...



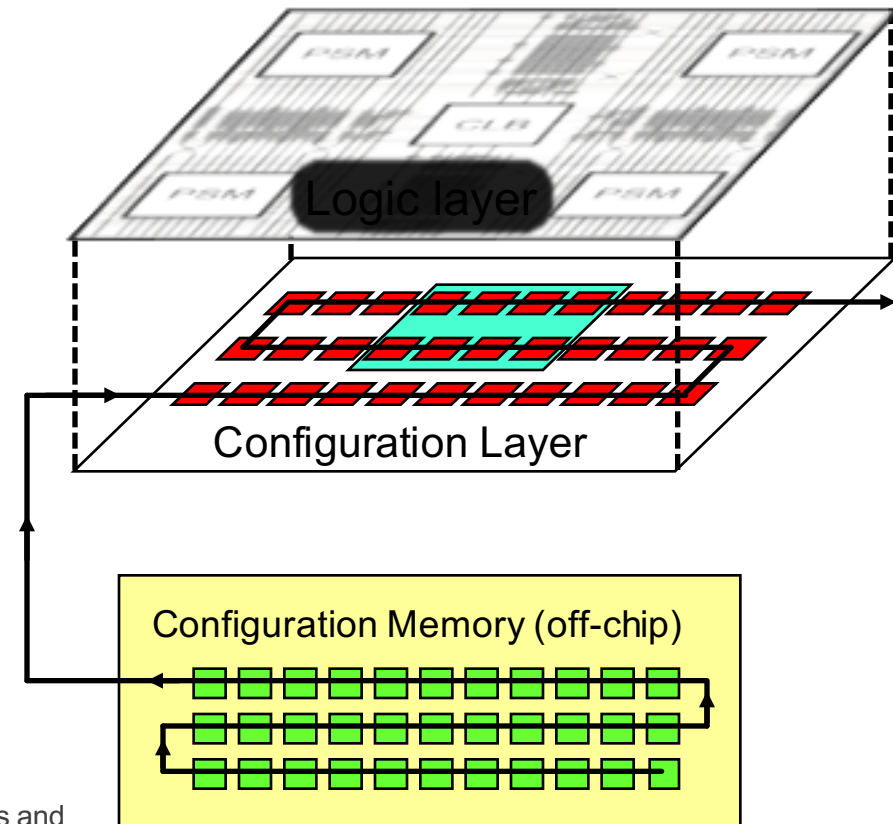
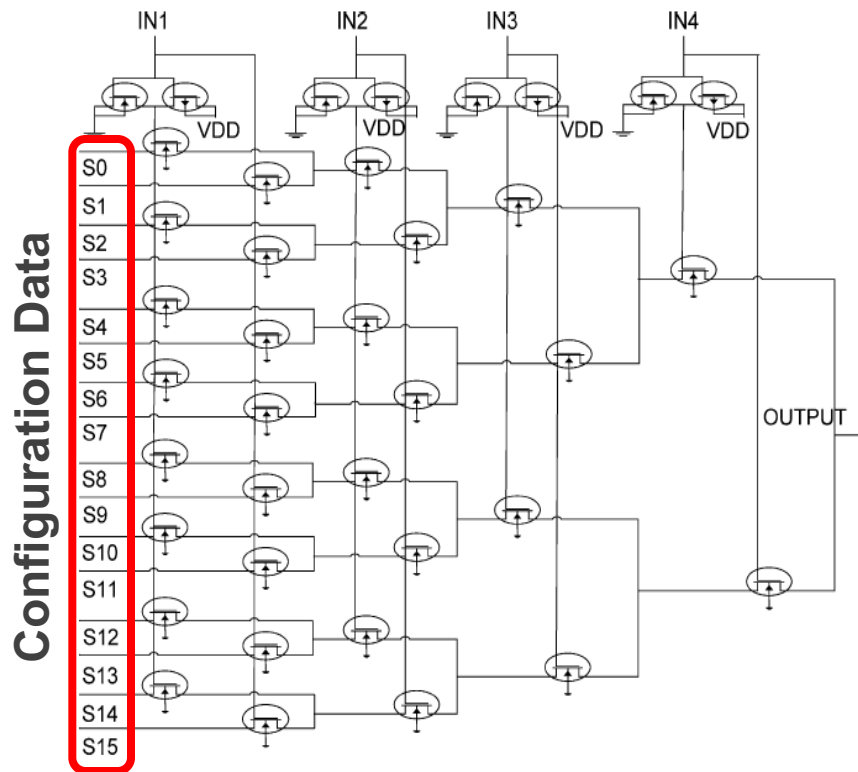
src: Xilinx Data Sheets

# Fine-Grained Reconfigurable Fabric

**Logic Layer:** Perform Computations

**Configuration Layer:** Which Computations to Perform

**Partial Reconfiguration:** Only parts of the Fabric are reconfigured



src: Kalenteridis et al. "A complete platform and toolset for system implementation on fine-grained reconfigurable hardware", Microprocessors and Microsystems 2004

# Coarse-Grained Reconfigurable Fabric

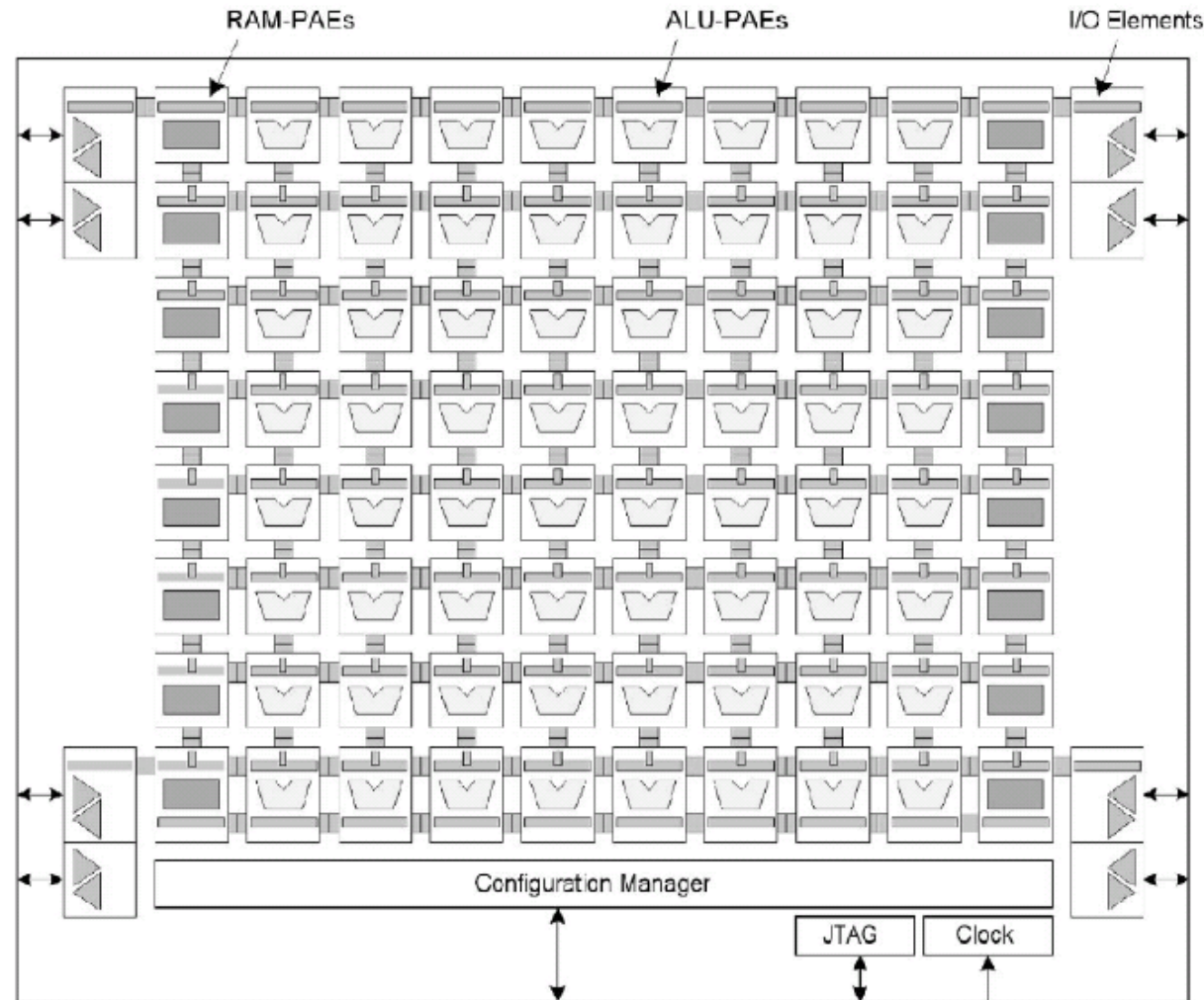
## Pros:

- Reduced Area
- Higher Frequency

Reduced  
reconfiguration  
latency

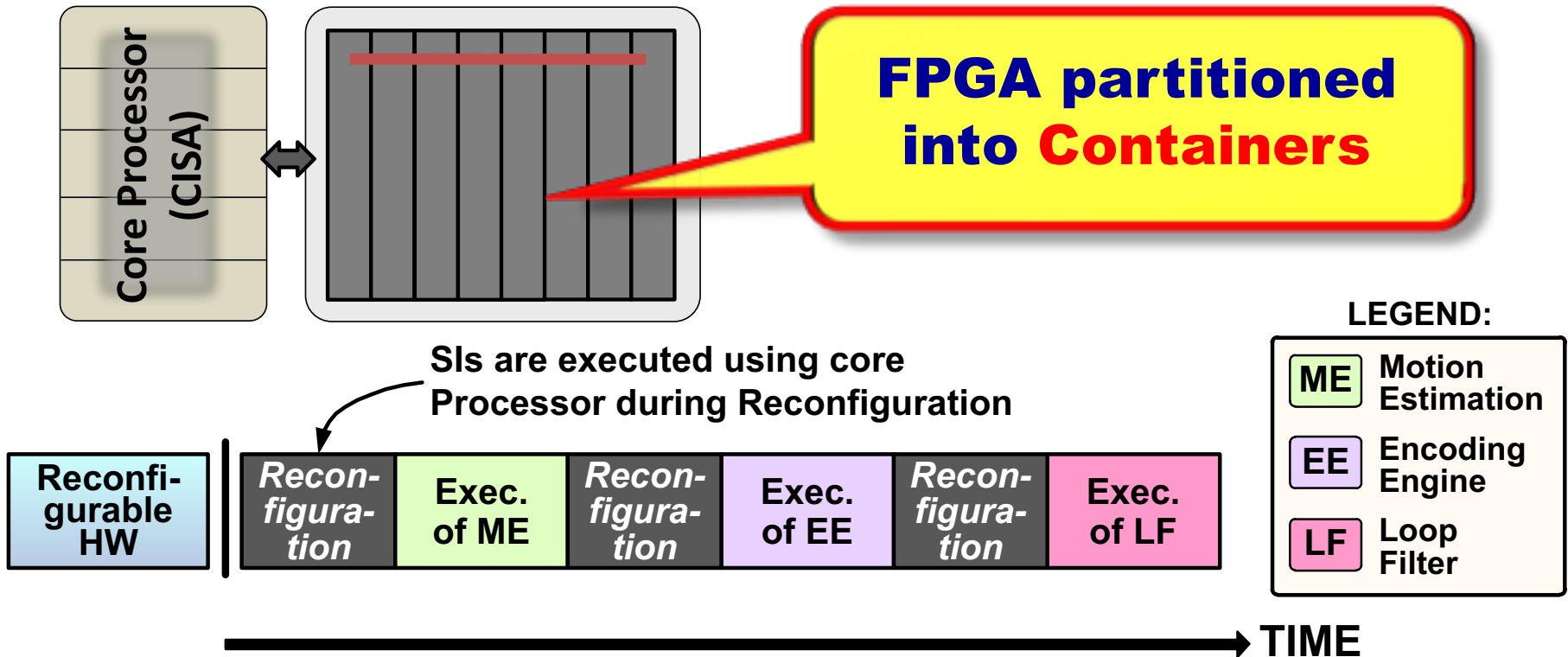
## Cons

- Reduced Flexibility
- Area Wastage  
→ In case  
computations do not  
suite the fabric (e.g.  
bit-level operations,  
bit shuffling)



src: PACT's XPP 64-A1 architecture

# Reconfigurable Computing: an example



## Runtime reconfigurable processor:

- + Efficient use of hardware: through high degree of parallelism, multiplexed used of reconf. fabric => energy efficient
- Reconfiguration Latency and Energy

# Memory



© Springer, 2010

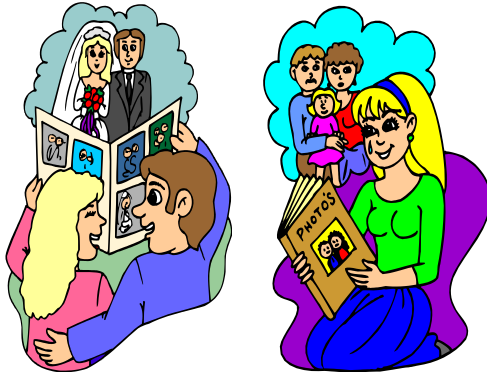
These slides use Microsoft clip arts. Microsoft copyright restrictions apply.



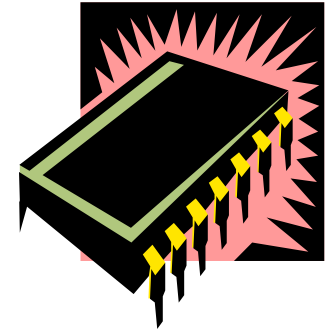
# Memory

---

Memories?



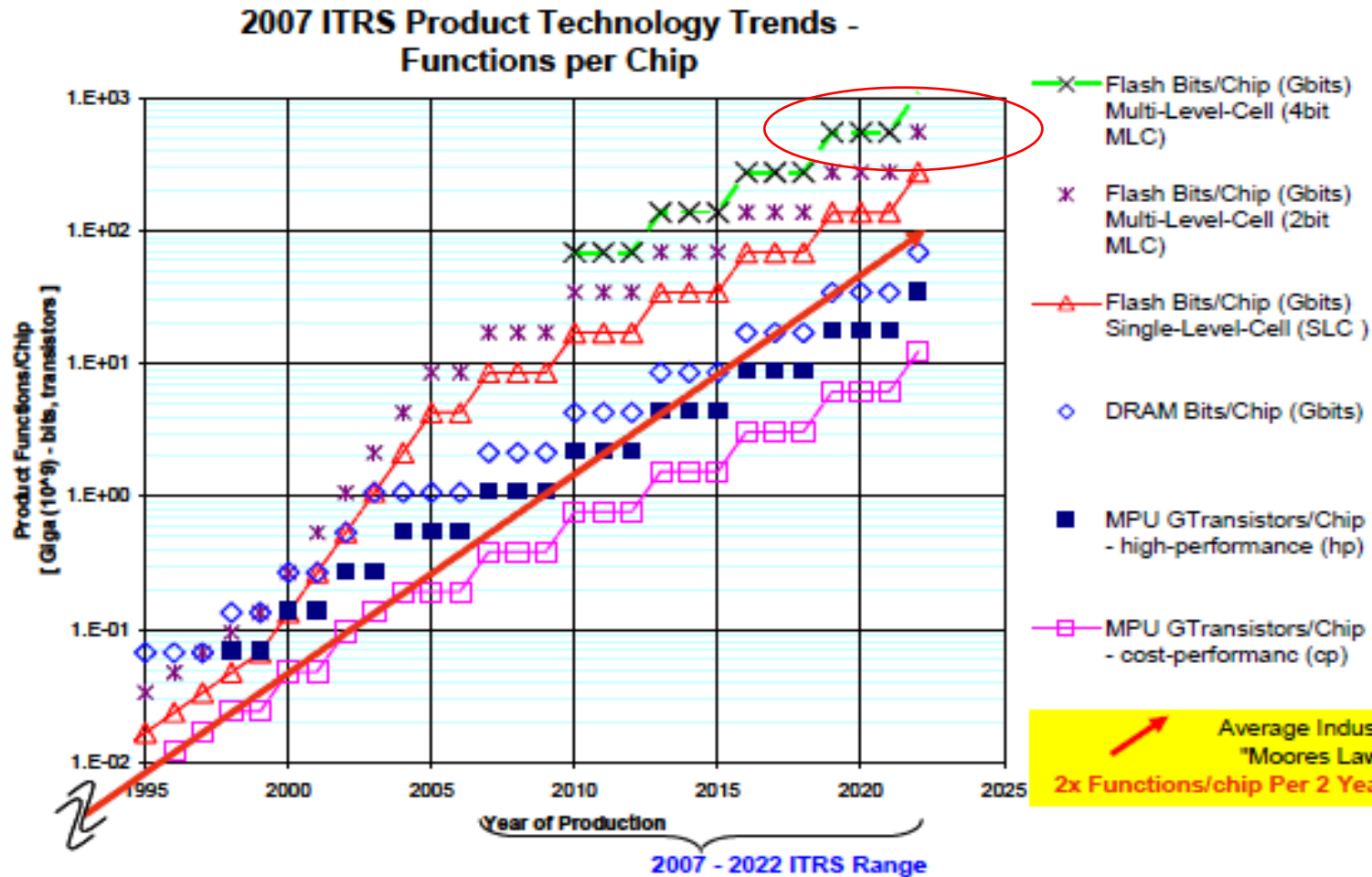
Oops!  
Memories!



For the memory, efficiency is again a concern:

- capacity
- energy efficiency
- speed (latency and throughput); predictable timing
- size
- cost
- other attributes (volatile vs. persistent, etc)

# Memory capacities expected to keep increasing



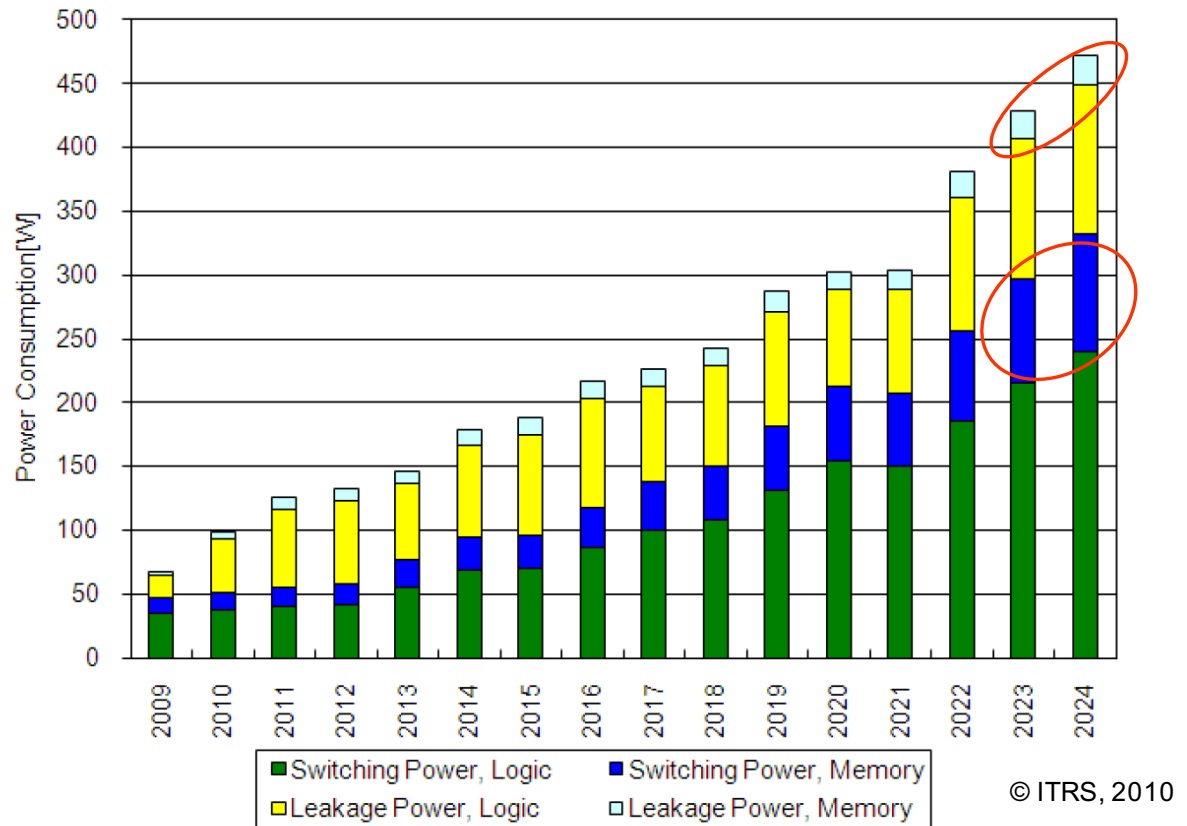
*Figure ORTC2 ITRS Product Function Size Trends:  
MPU Logic Gate Size (4-transistor); Memory Cell Size [SRAM (6-transistor); Flash (SLC and MLC), and  
DRAM (transistor + capacitor)]--Updated*

[ITRS Update 2008]

# Where is the power consumed?

## - Stationary systems -

- According to *International Technology Roadmap for Semi-conductors* (ITRS), 2010 update, [www.itrs.net]

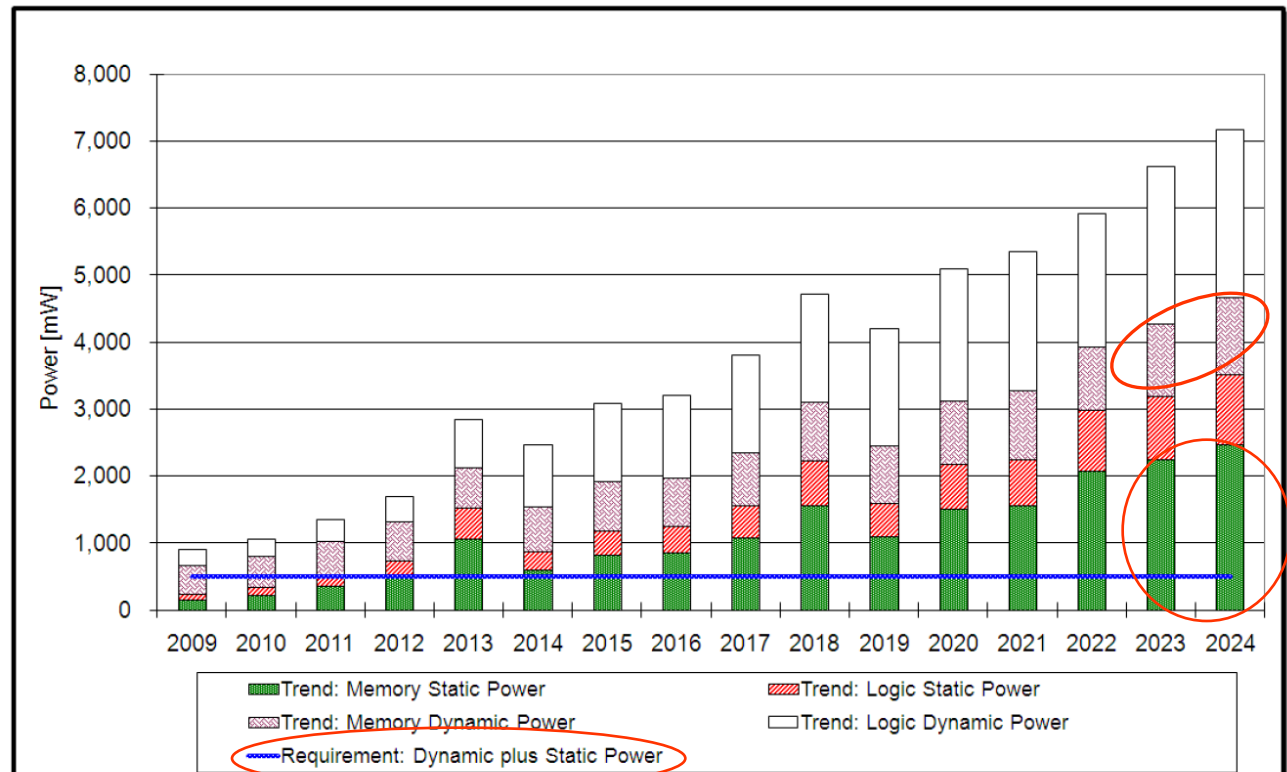


- Switching (dynamic) power, logic dominating
- Overall power consumption a nightmare for environmentalists

# Where is the power consumed?

## - Consumer portable systems -

- According to *International Technology Roadmap for Semiconductors* (ITRS), 2010 update, [www.itrs.net]
- Based on current trends

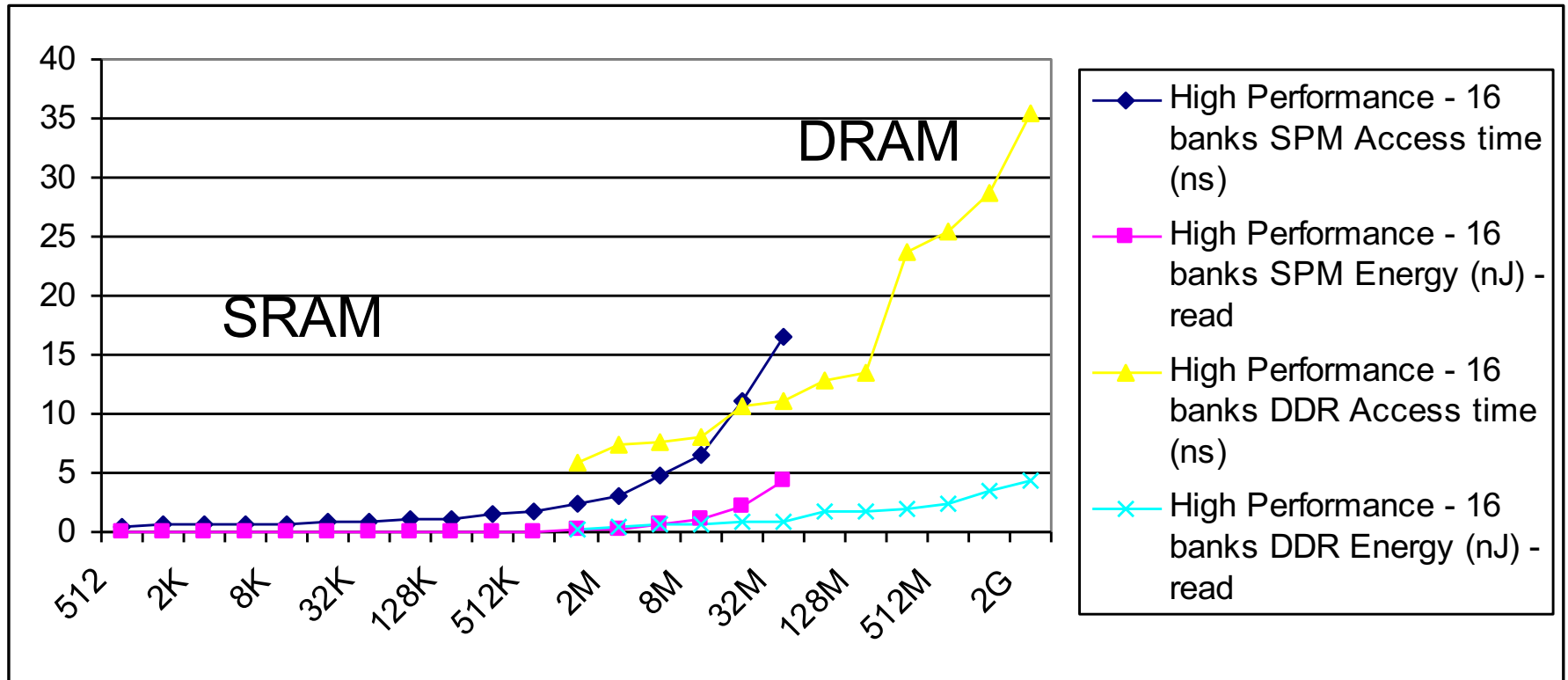


© ITRS, 2010

- Memory and logic, static and dynamic relevant

# Energy consumption and access times of memories

Example CACTI: Scratchpad (SRAM) vs. DRAM (DDR2):

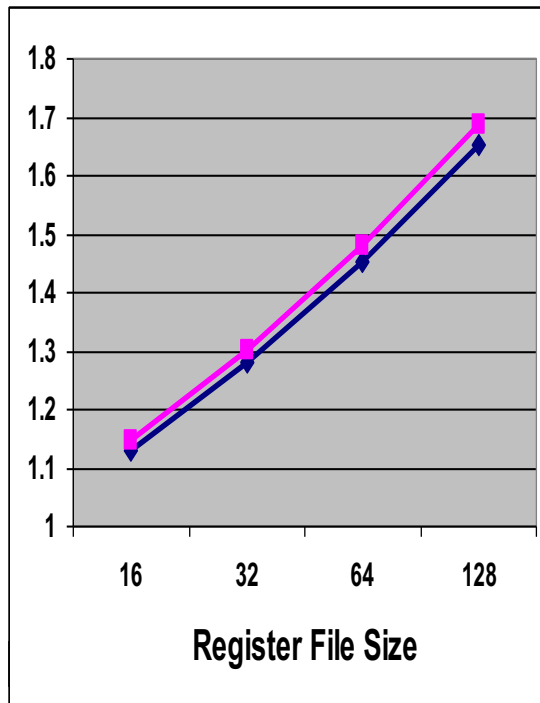


16 bit read; size in bytes;  
65 nm for SRAM, 80 nm for DRAM

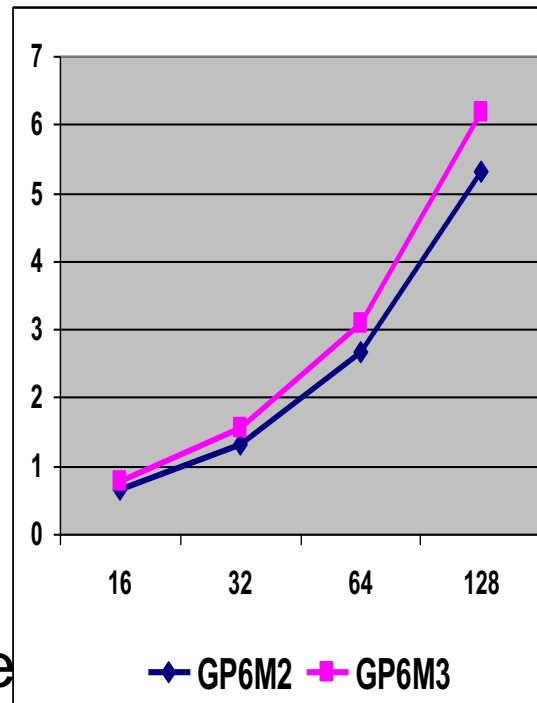
Source: Olivera Jovanovic,  
TU Dortmund, 2011

# Access times and energy consumption for multi-ported register files

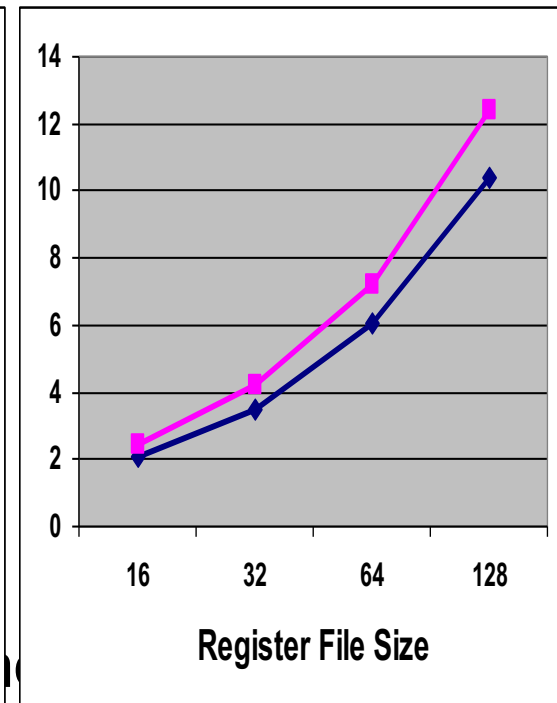
Cycle Time (ns)



Area ( $\lambda^2 \times 10^6$ )



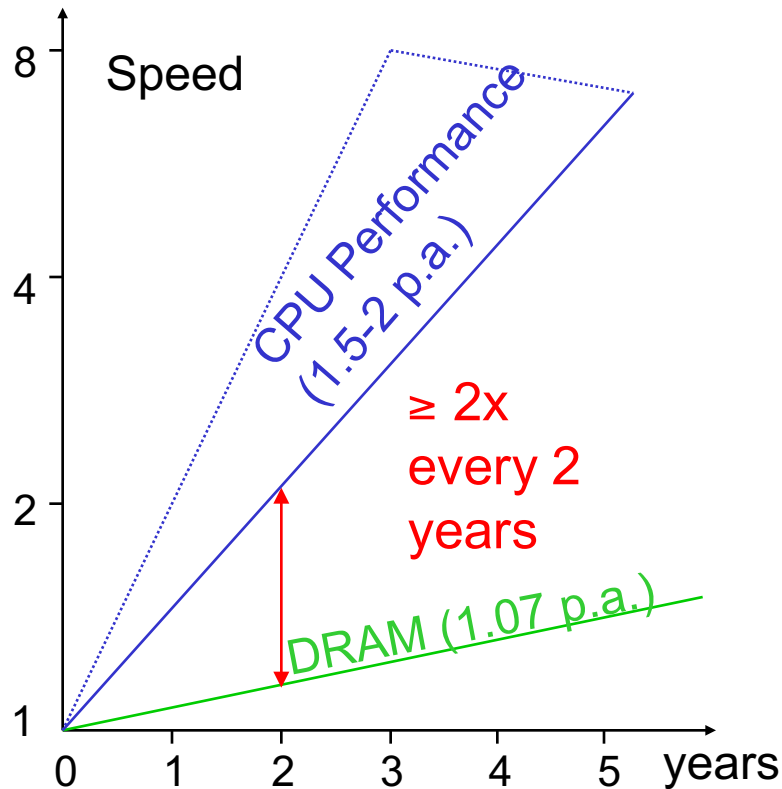
Power (W)



Source and © H. Valero, 2001

# Trends for the Speeds

Speed gap between processor and main DRAM increases



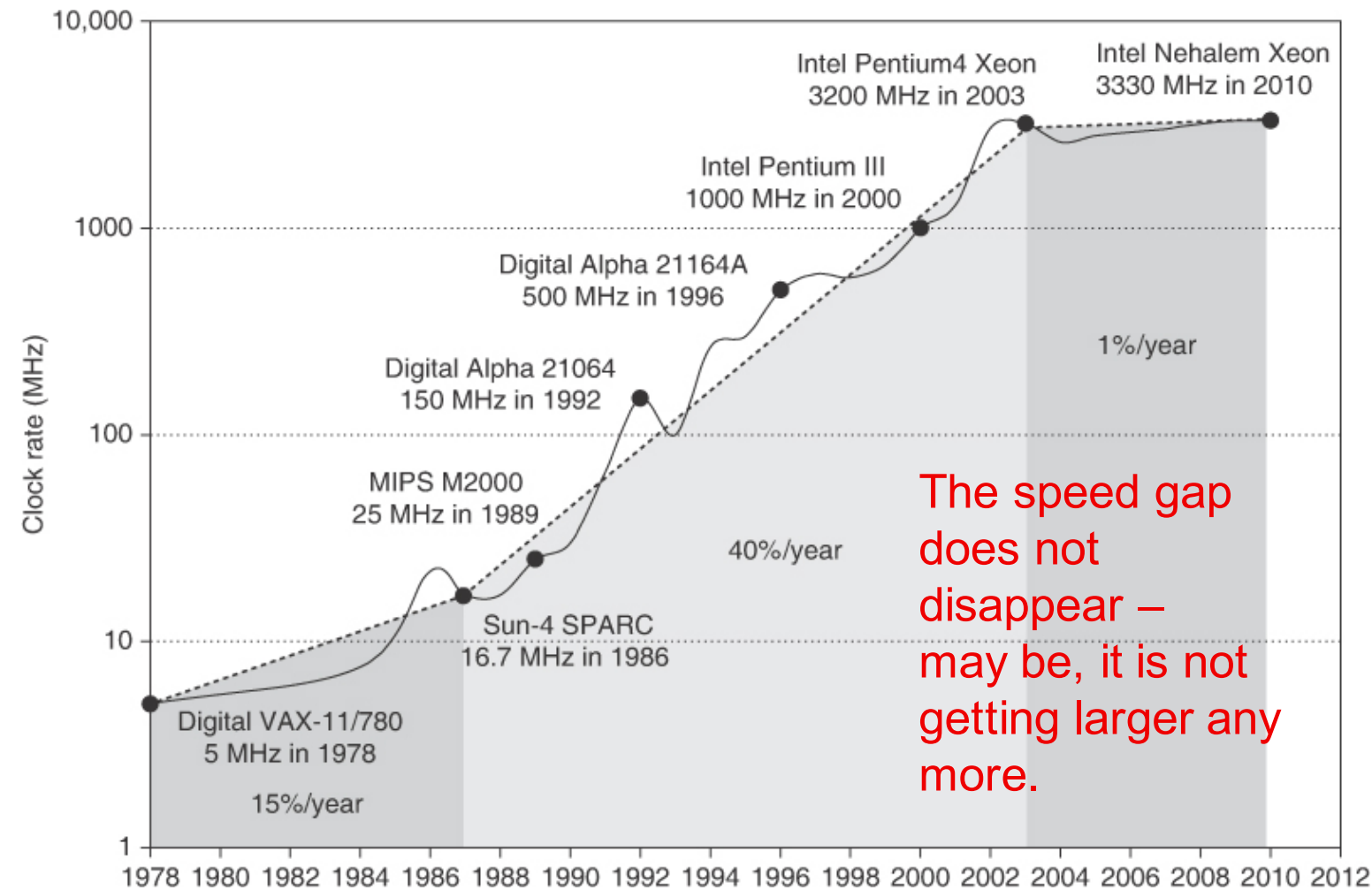
Similar problems also for embedded systems & MPSoCs

- ➡ Memory access times >> processor cycle times
- ➡ “Memory wall” problem



[P. Machanik: Approaches to Addressing the Memory Wall, TR Nov. 2002, U. Brisbane]

# However, clock speed increases have come to a halt

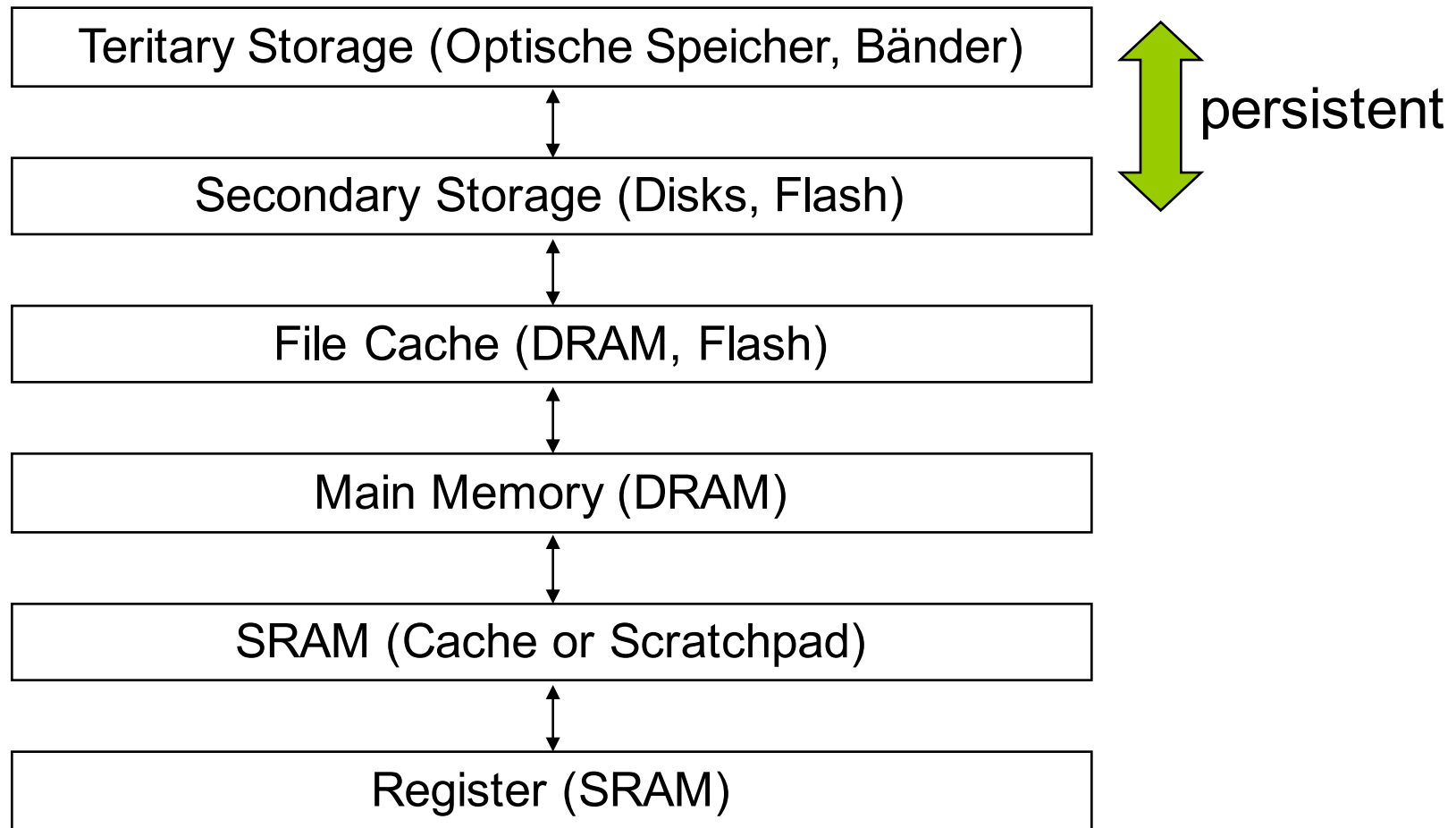


Copyright © 2011, Elsevier Inc. All rights Reserved.

[Hennessy/Patterson: Computer Architecture, 5th ed., 2011]



# Memory Hierarchy



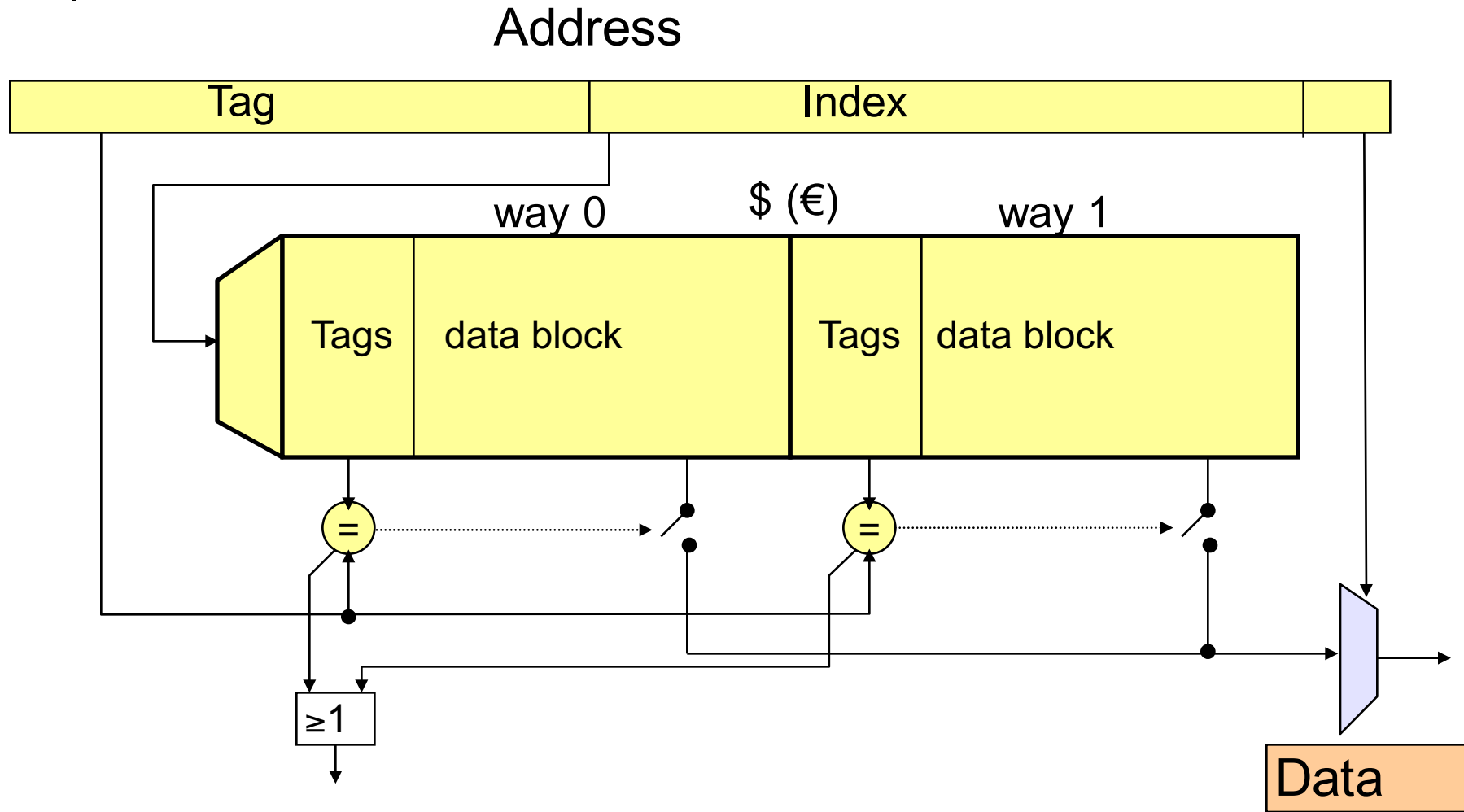
# Recover Your Knowledge on Cache

---

- Fully-associative mapping, direct mapping, set-associative mapping
- Cache replacement policy
- Temporal locality and spatial locality

# Set-associative cache $n$ -way cache

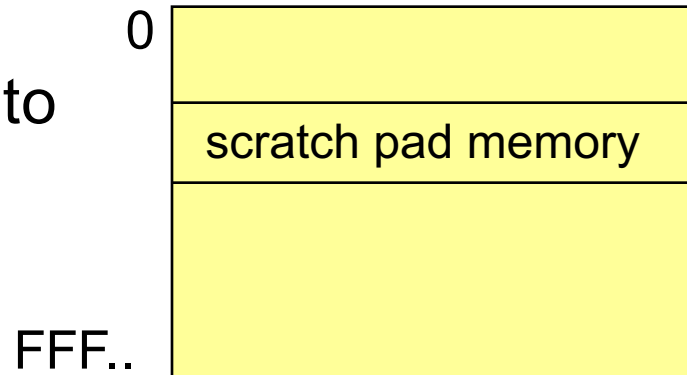
$|\text{Set}| = 2$



# Hierarchical memories using scratch pad memories (SPM)

SPM is a small,  
physically separate  
memory mapped into  
the address space

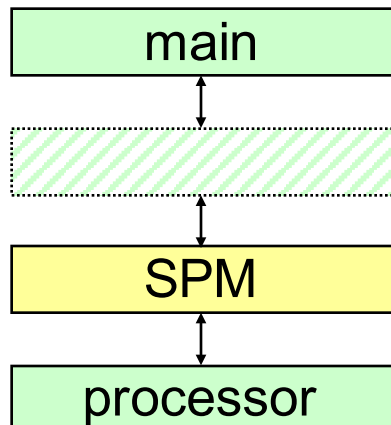
Address space



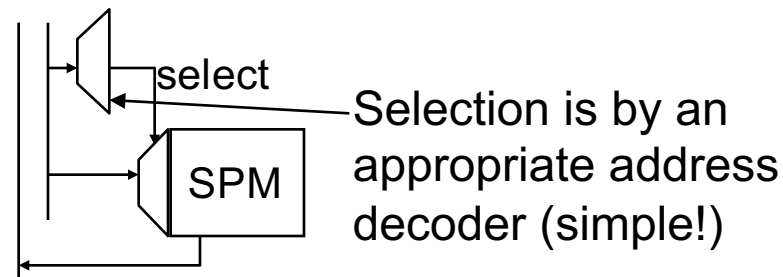
Examples:

- Most ARM cores allow tightly coupled memories
- IBM Cell
- Infineon TriCore
- Many multi-cores, due to high costs of coherent caches

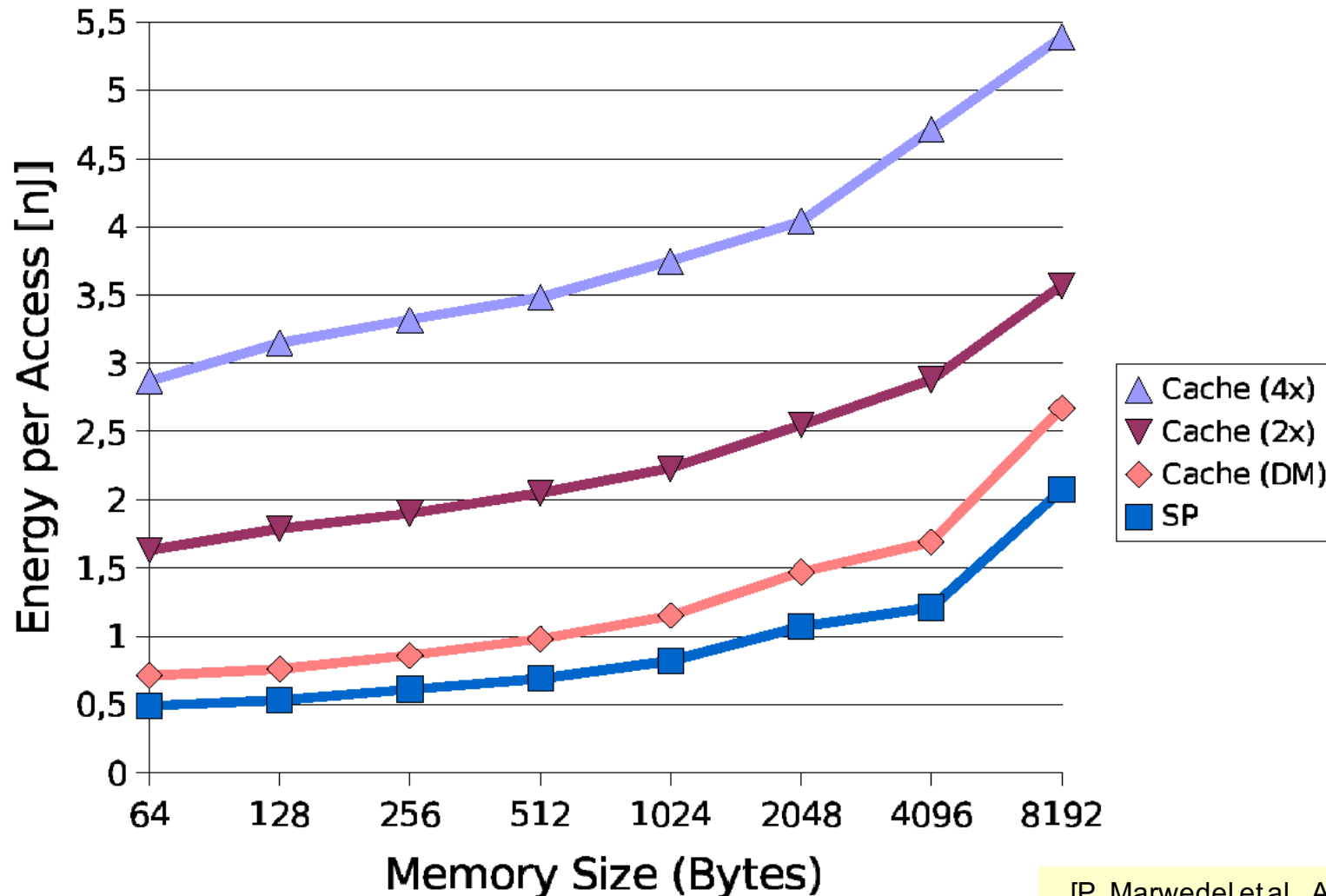
Hierarchy



no tag memory



# Influence of the associativity



[P. Marwedel et al., ASPDAC, 2004]

# Summary

---

- Processing
  - VLIW/EPIC processors
  - MPSoCs
- FPGAs
- Memories
  - “Small is beautiful”  
(in terms of energy consumption, access times, size)

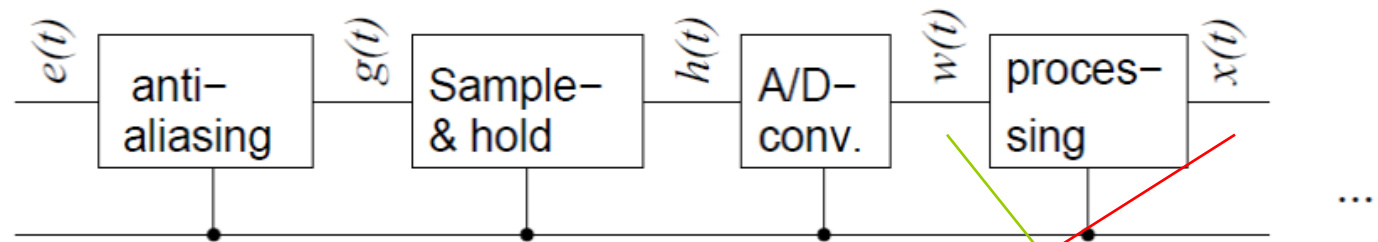
# Spare Slides

---

# Key requirement #3: Run-time efficiency

## - Domain-oriented architectures -

Example: Filtering in Digital signal processing (DSP)

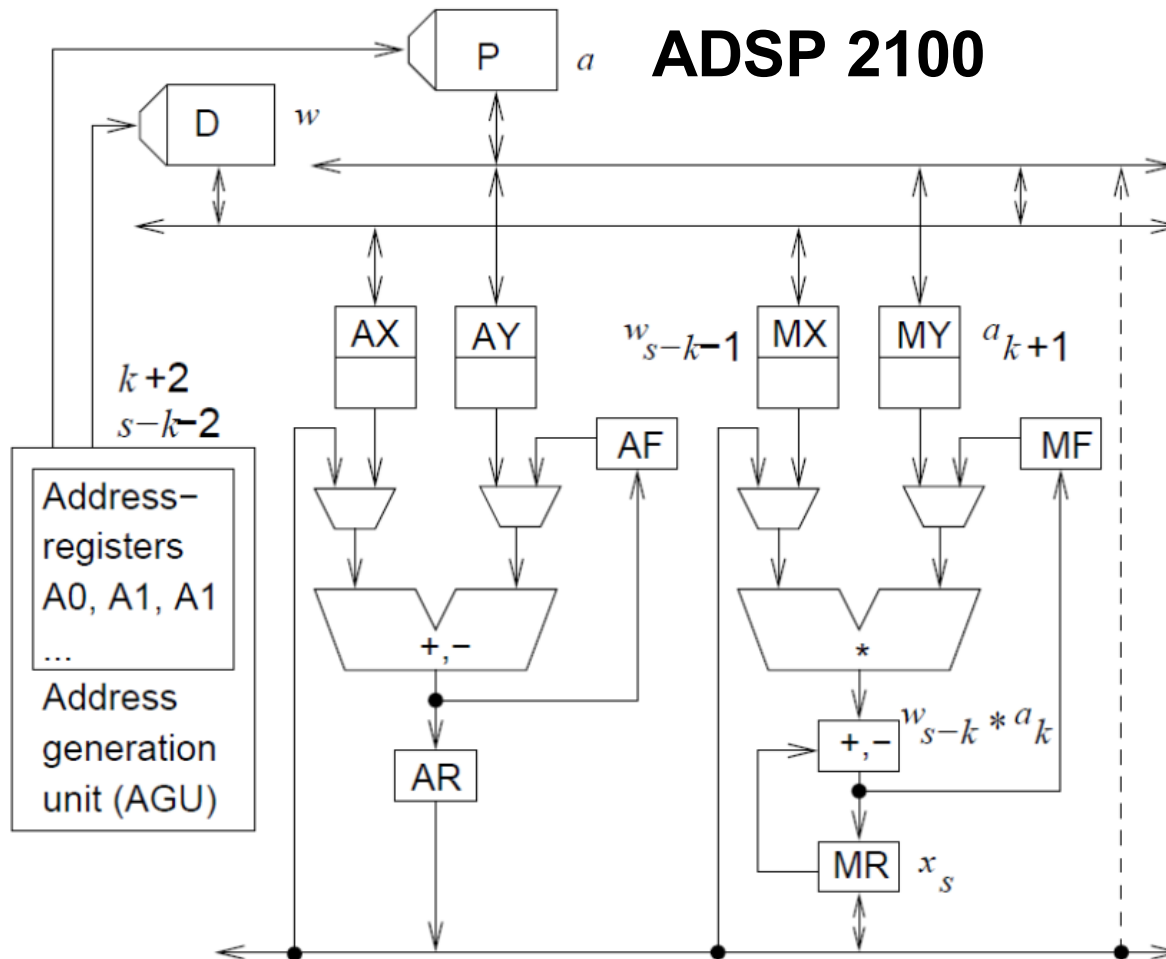


$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

Signal at  $t=t_s$  (sampling points)



# Filtering in digital signal processing



$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

```
-- outer loop over
-- sampling times  $t_s$ 
{ MR:=0; A1:=1; A2:=s-1;
  MX:=w[s]; MY:=a[0];
  for (k=0; k <= (n-1); k++)
  { MR:=MR + MX * MY;
    MX:=w[A2]; MY:=a[A1];
    A1++; A2--;
  }
  x[s]:=MR;
}
```

Maps nicely

# DSP-Processors: multiply/accumulate (MAC) and zero-overhead loop (ZOL) instructions

---

```
MR:=0; A1:=1; A2:=s-1; MX:=w[s]; MY:=a[0];
```

```
for ( k:=0 <= n-1)
```

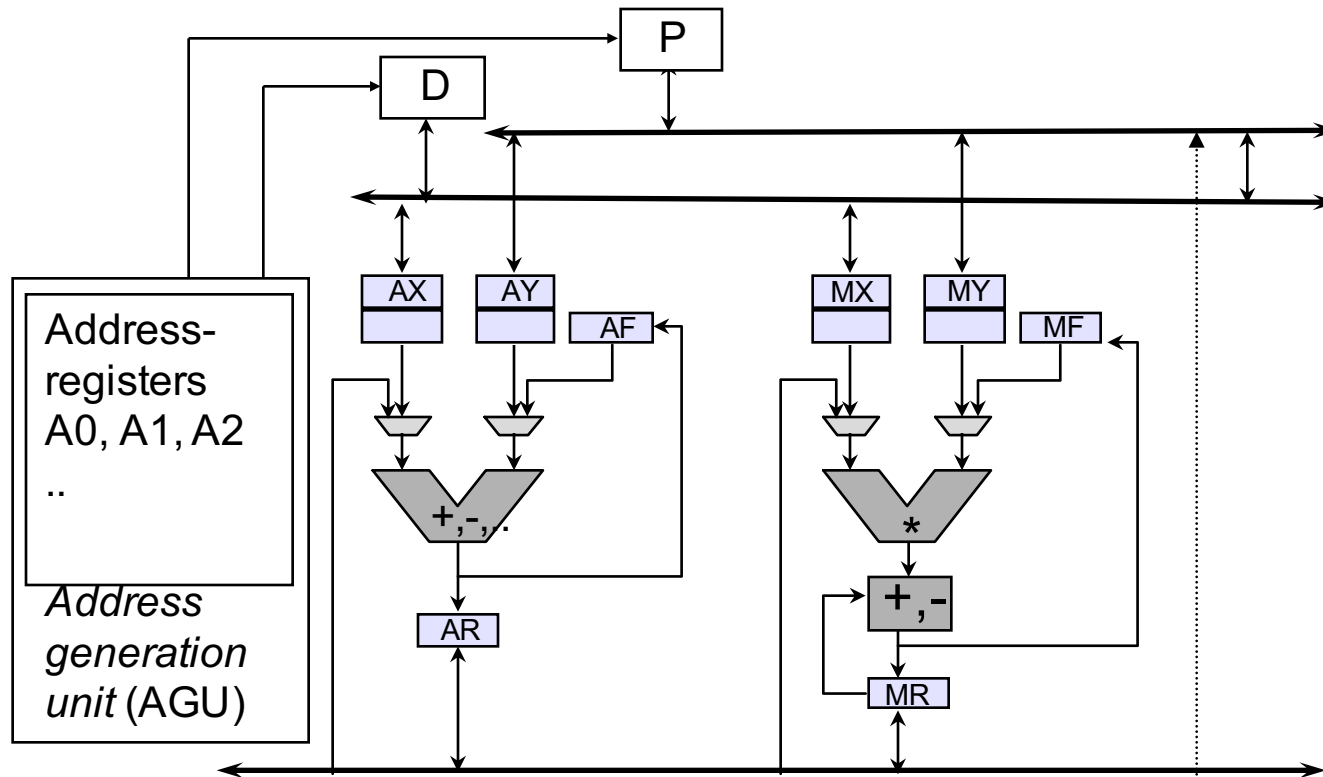
```
{MR:=MR+MX*MY; MY:=a[A1]; MX:=w[A2]; A1++; A2--}
```

Multiply/accumulate (MAC) instruction

Zero-overhead loop (ZOL) instruction preceding MAC instruction.  
Loop testing done in parallel to MAC operations.

# Heterogeneous registers

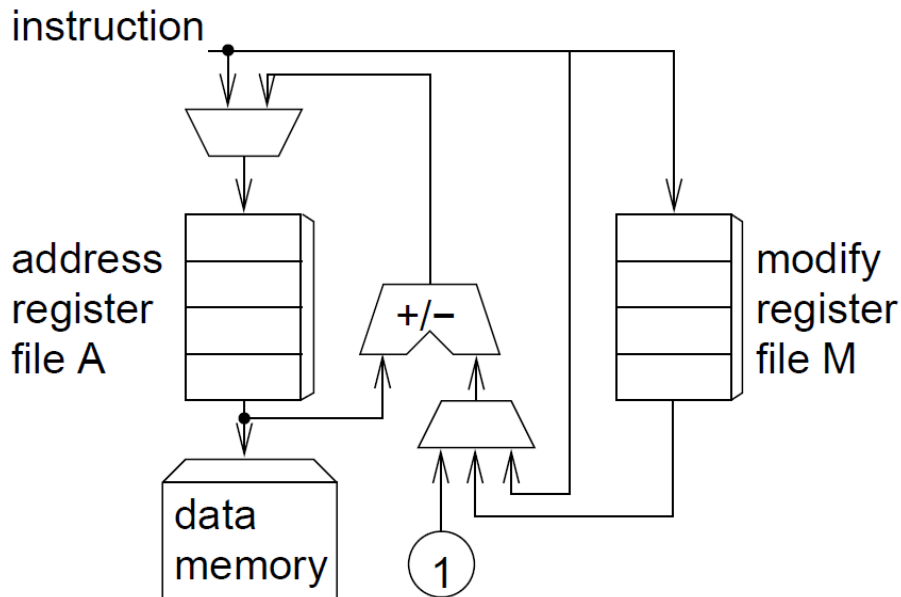
## Example (ADSP 210x):



## Different functionality of registers An, AX, AY, AF, MX, MY, MF, MR

# Separate address generation units (AGUs)

## Example (ADSP 210x):

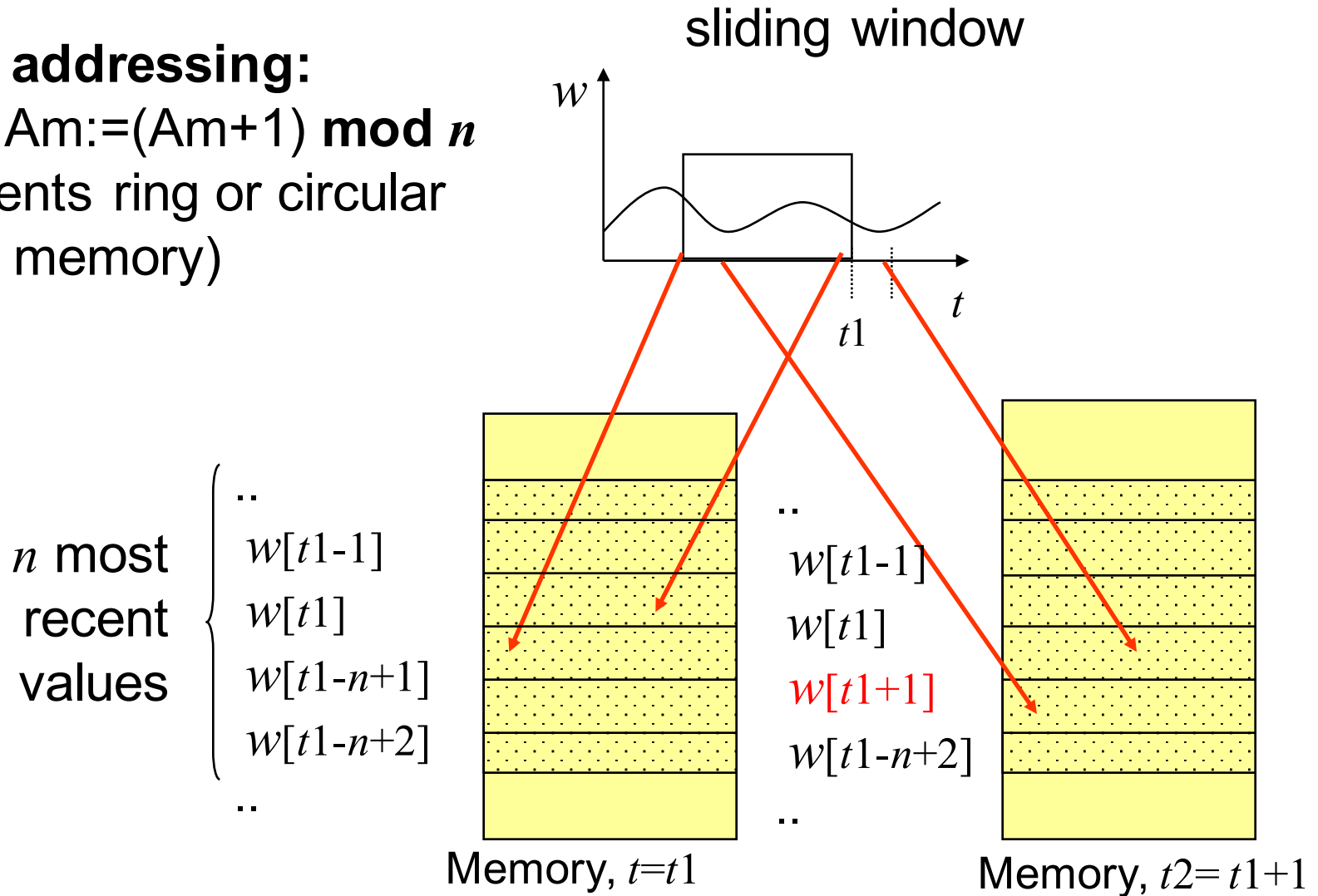


- Data memory can only be fetched with address contained in A,
  - but this can be done in parallel with operation in main data path (takes effectively 0 time).
  - $A := A \pm 1$  also takes 0 time,
  - same for  $A := A \pm M$ ;
  - $A := \text{<immediate in instruction>}$  requires extra instruction
- ☞ Minimize load immediates
- ☞ Optimization in optimization chapter

# Modulo addressing

## Modulo addressing:

$A_{m++} \equiv A_m := (A_m + 1) \bmod n$   
(implements ring or circular buffer in memory)



# Cycles/access as a function of the size of the list

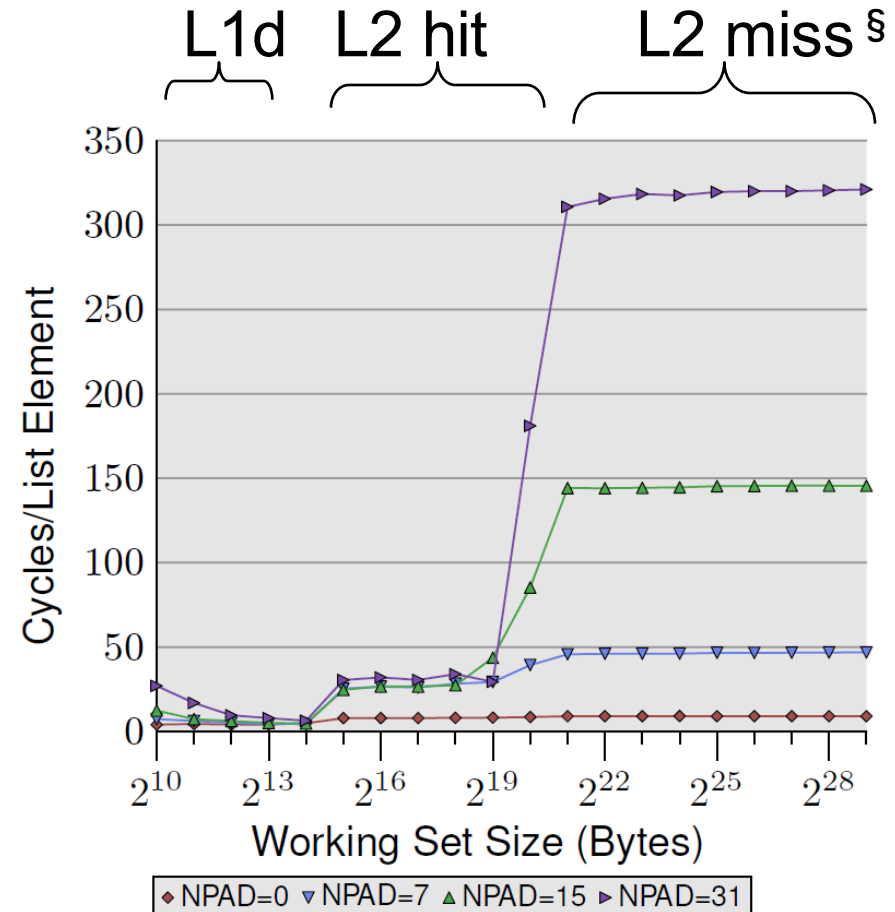
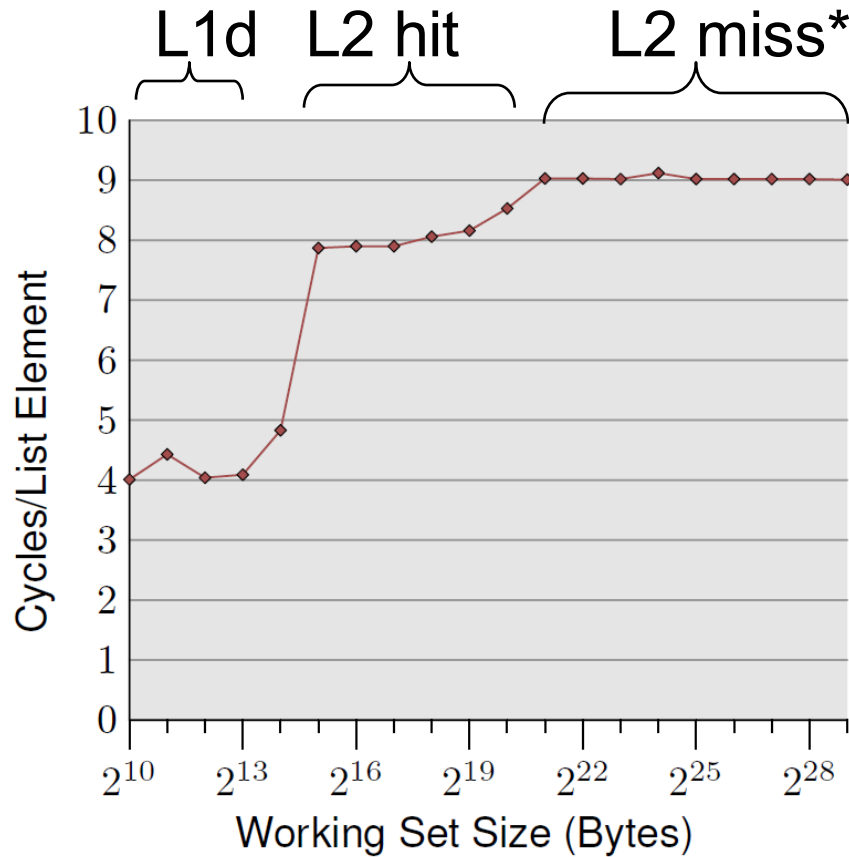


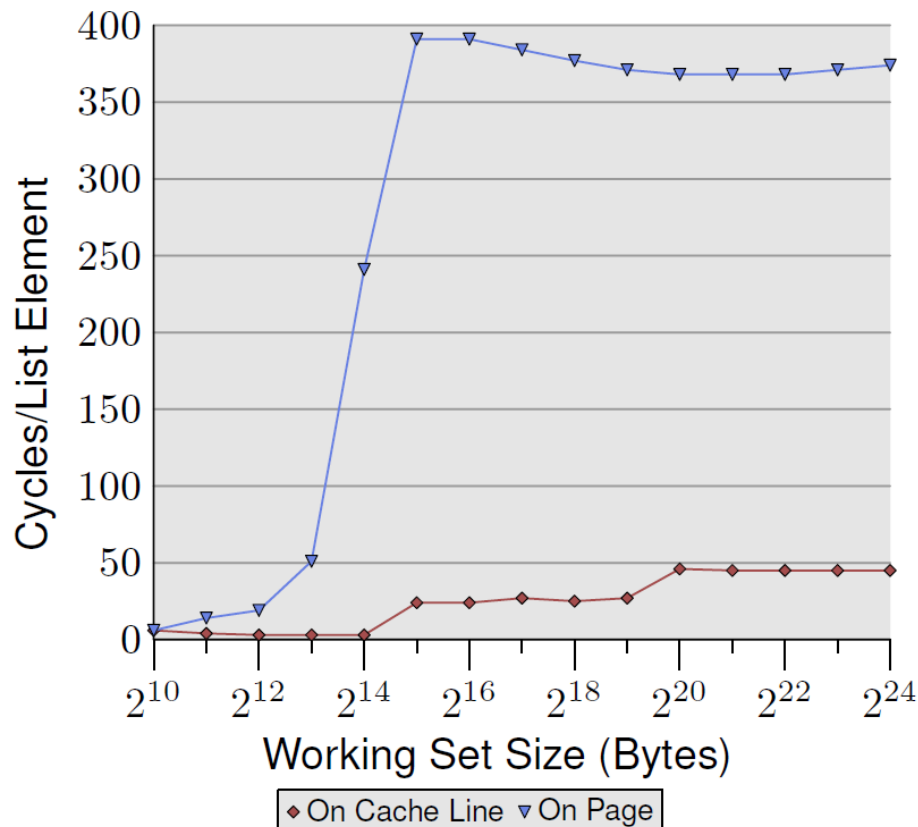
Figure 3.10: Sequential Read Access, NPAD=0

\* prefetching succeeds

§ prefetching fails

# Impact of TLB misses and larger caches

Elements on different pages; run time increase when exceeding the size of the TLB



Larger caches are shifting the steps to the right

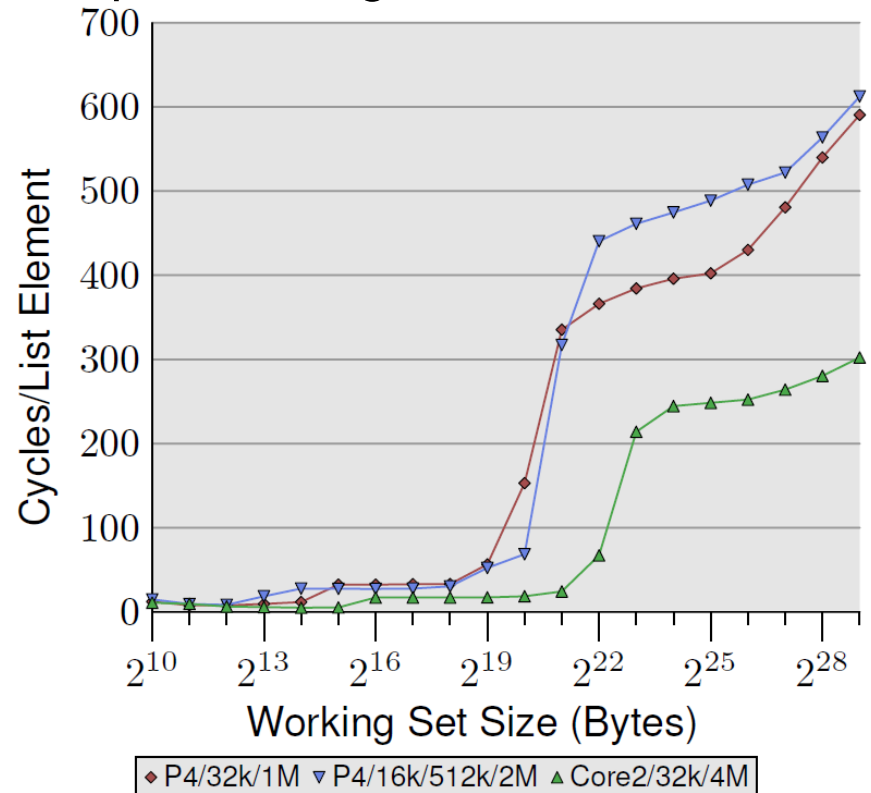
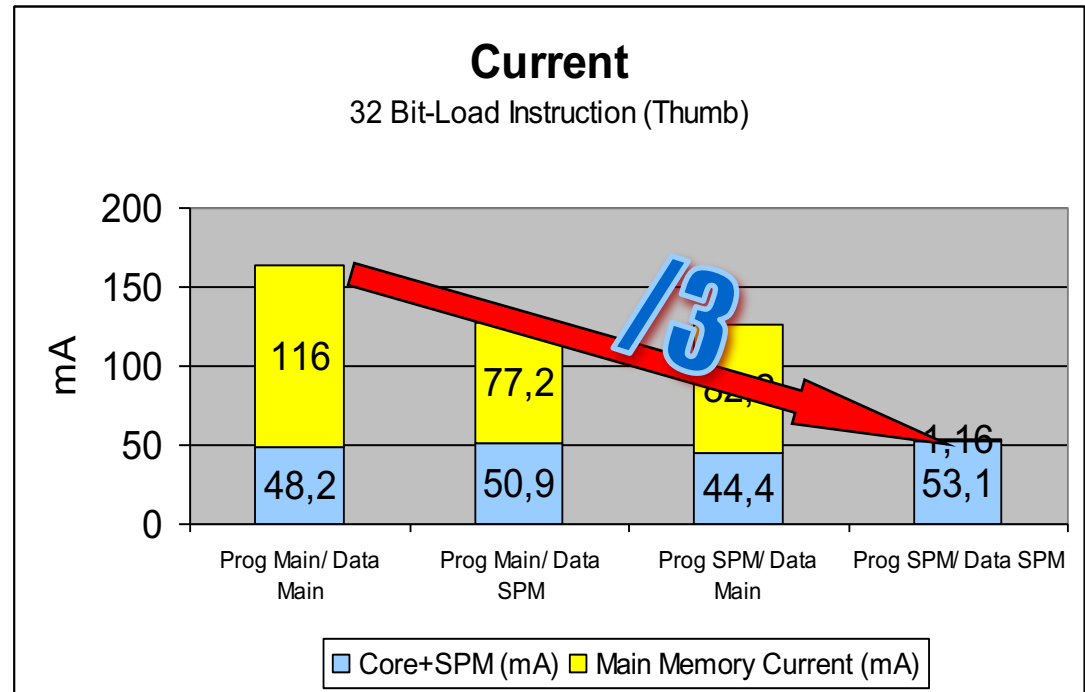
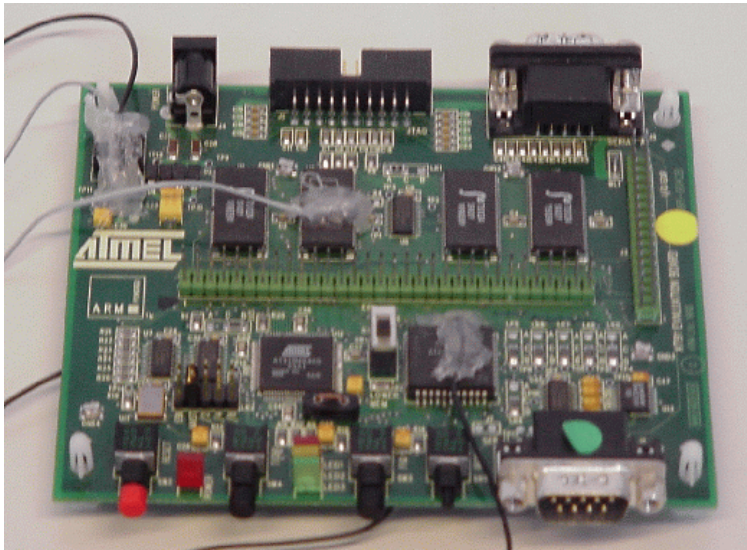


Figure 3.14: Advantage of Larger L2/L3 Caches

# Comparison of currents using measurements

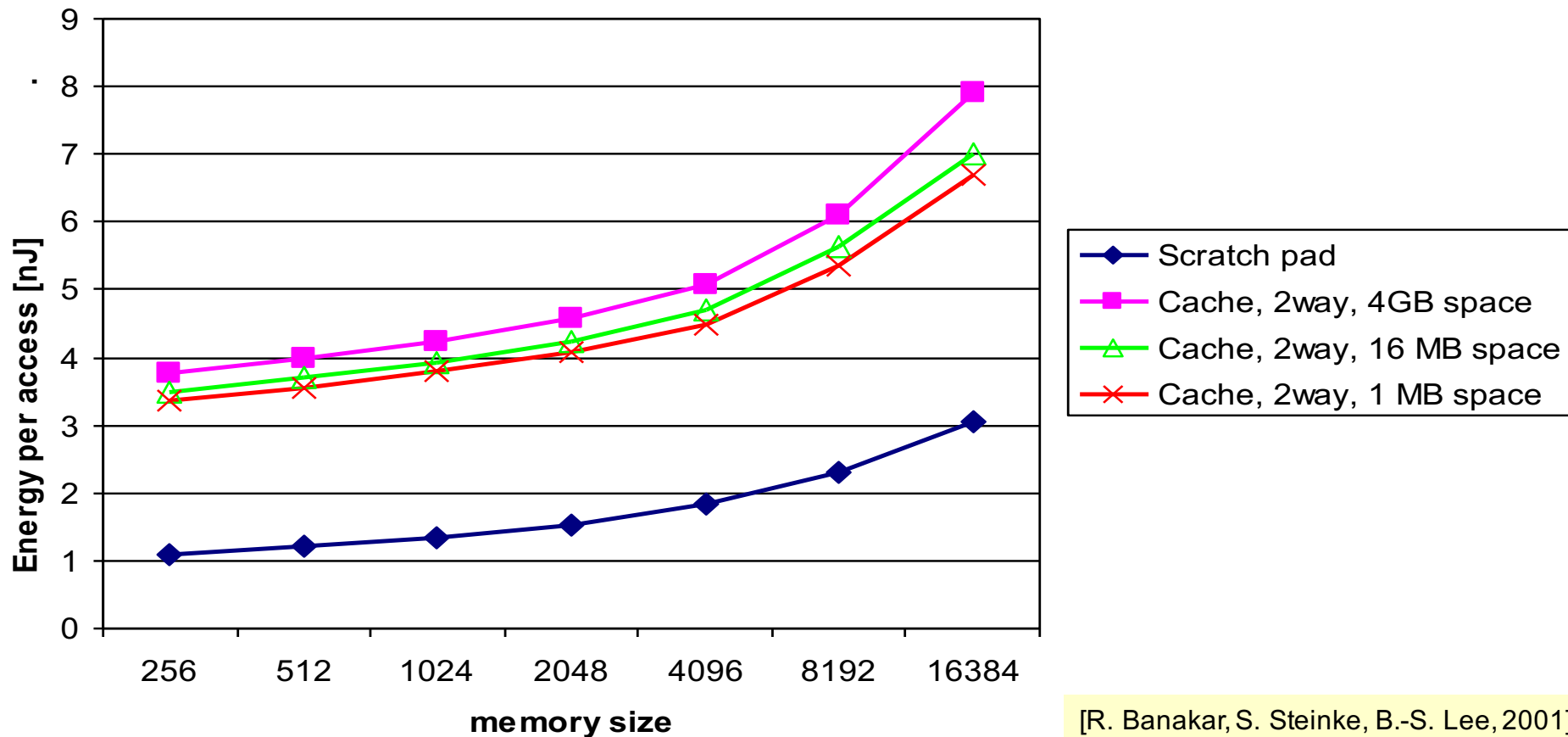
E.g.: ATMEL board with  
ARM7TDMI and  
ext. SRAM





# Why not just use a cache ?

## 2. Energy for parallel access of sets, in comparators, muxes.



# Influence of the associativity

