

ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 12 **Memory Hierarchy Review**

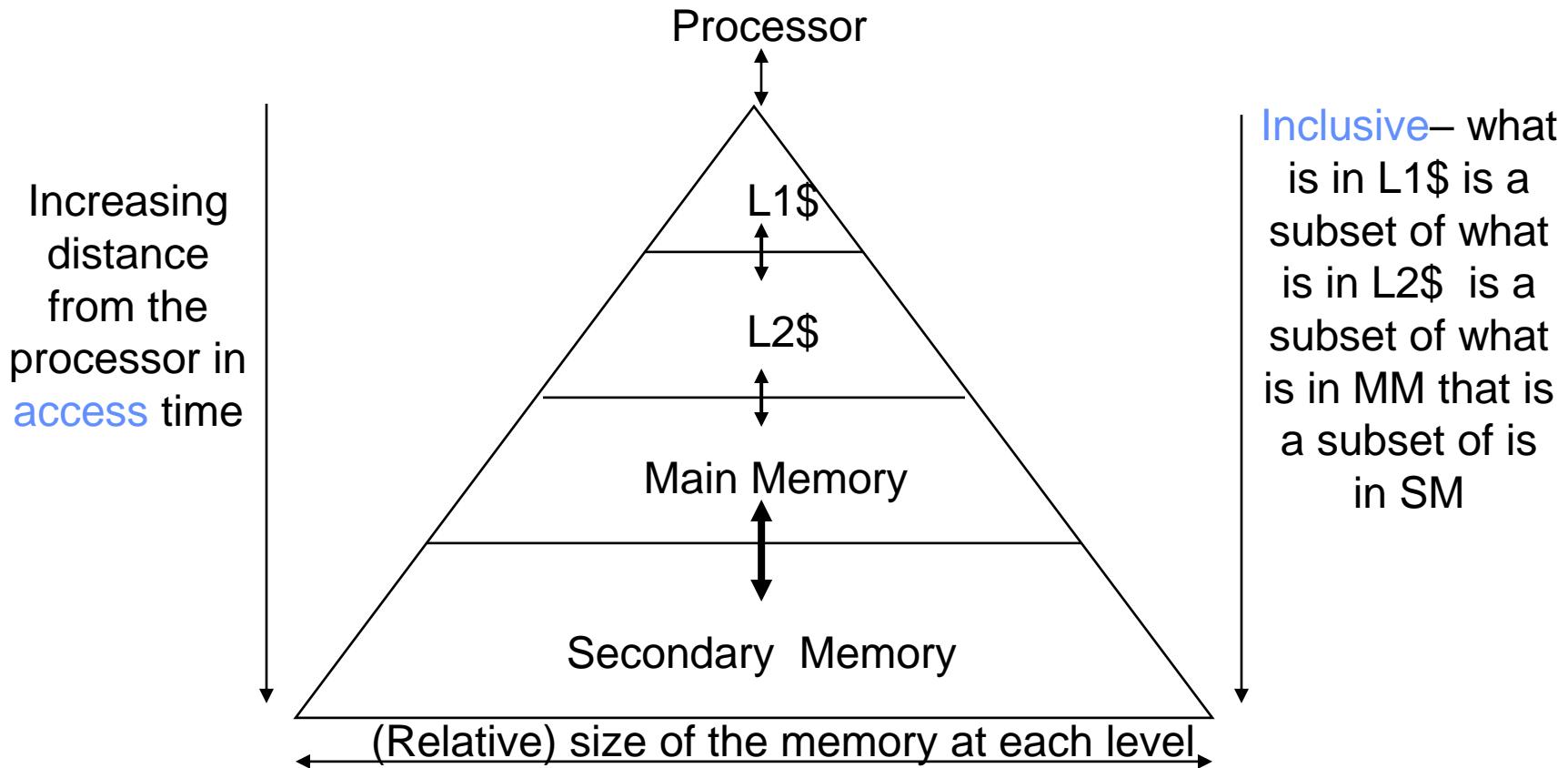
02/27/2023

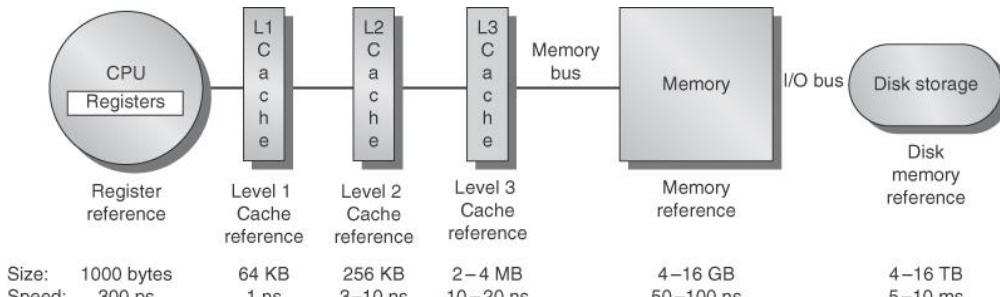
Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

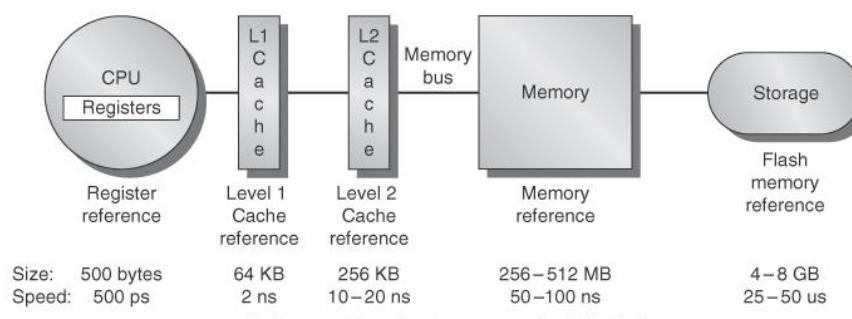
Review: The Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology





(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10⁹—from picoseconds to milliseconds—and that the size units change by a factor of 10¹²—from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

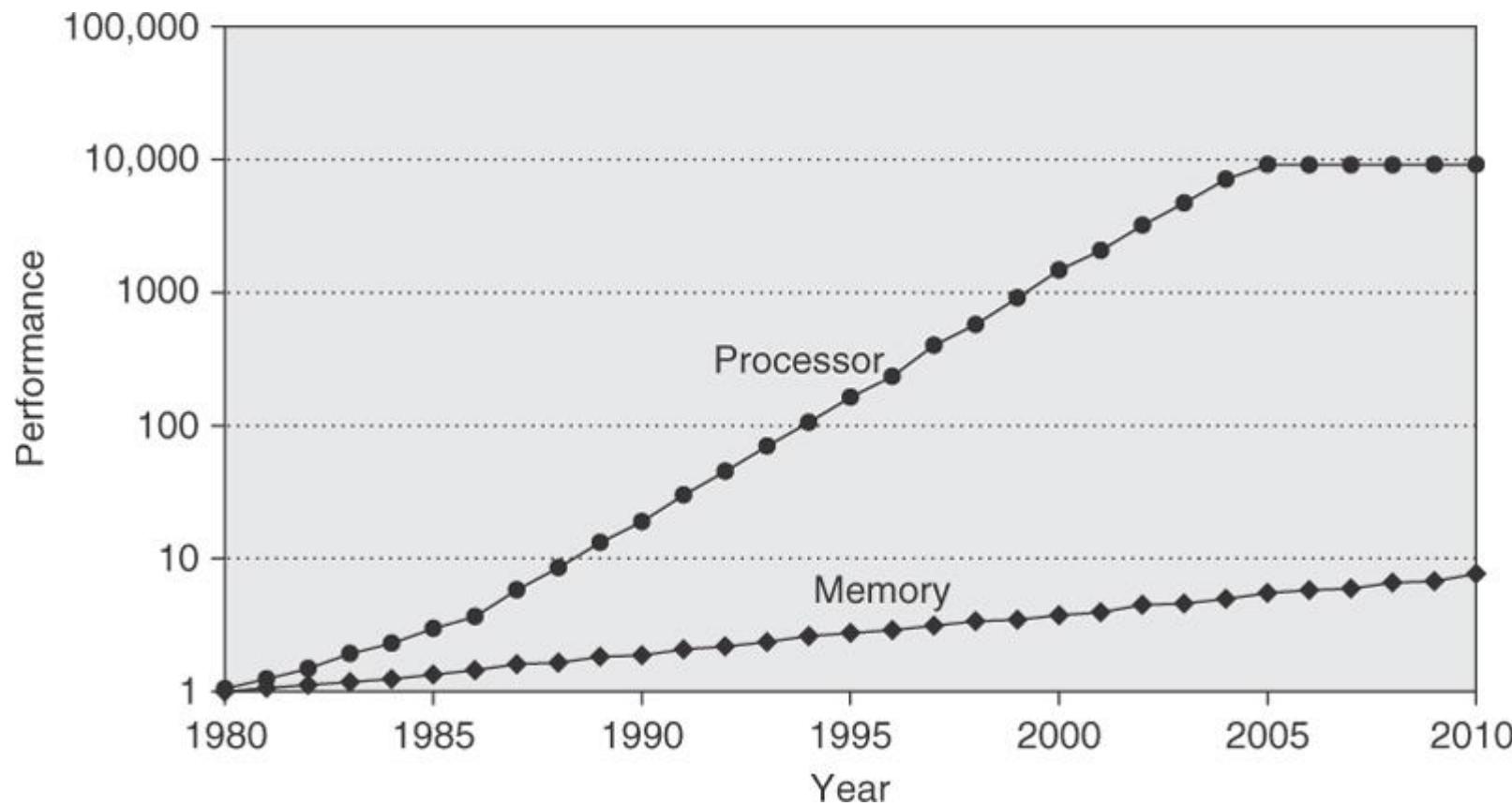
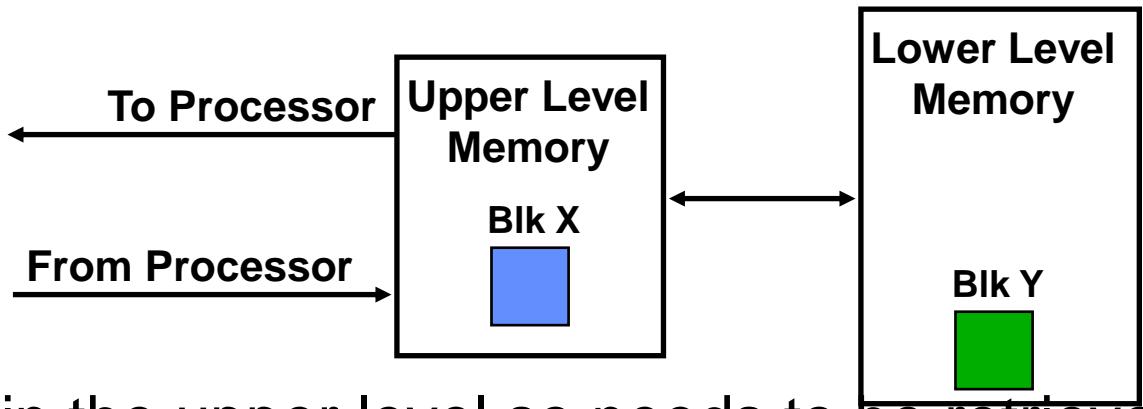


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

The Memory Hierarchy: Terminology

- **Hit:** data is in some block in the upper level (**Blk X**)
 - **Hit Rate:** the fraction of memory accesses found in the upper level
 - **Hit Time:** Time to access the upper level which consists of
RAM access time + Time to determine hit/miss



- **Miss:** data is not in the upper level so needs to be retrieved from a block in the lower level (**Blk Y**)
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty:** Time to replace a block in the upper level
+ Time to deliver the block the processor
 - **Hit Time** << **Miss Penalty**

How is the Hierarchy Managed?

- registers ↔ memory
 - by compiler (programmer?)
- cache ↔ main memory
 - [by the cache controller hardware](#)
- main memory ↔ disks
 - by the operating system (virtual memory)
 - virtual to physical address mapping assisted by the hardware ([TLB](#))
 - by the programmer (files)

Cache

- Two questions to answer (in hardware):
 - Q1: How do we know if a data item is in the cache?
 - Q2: If it is, how do we find it?
- Direct mapped
 - For each item of data at the lower level, there is exactly one location in the cache where it might be - so lots of items at the lower level must **share** locations in the upper level
 - Address mapping:
(block address) modulo (# of blocks in the cache)
 - First consider block sizes of **one word**

Caching: A Simple First Example

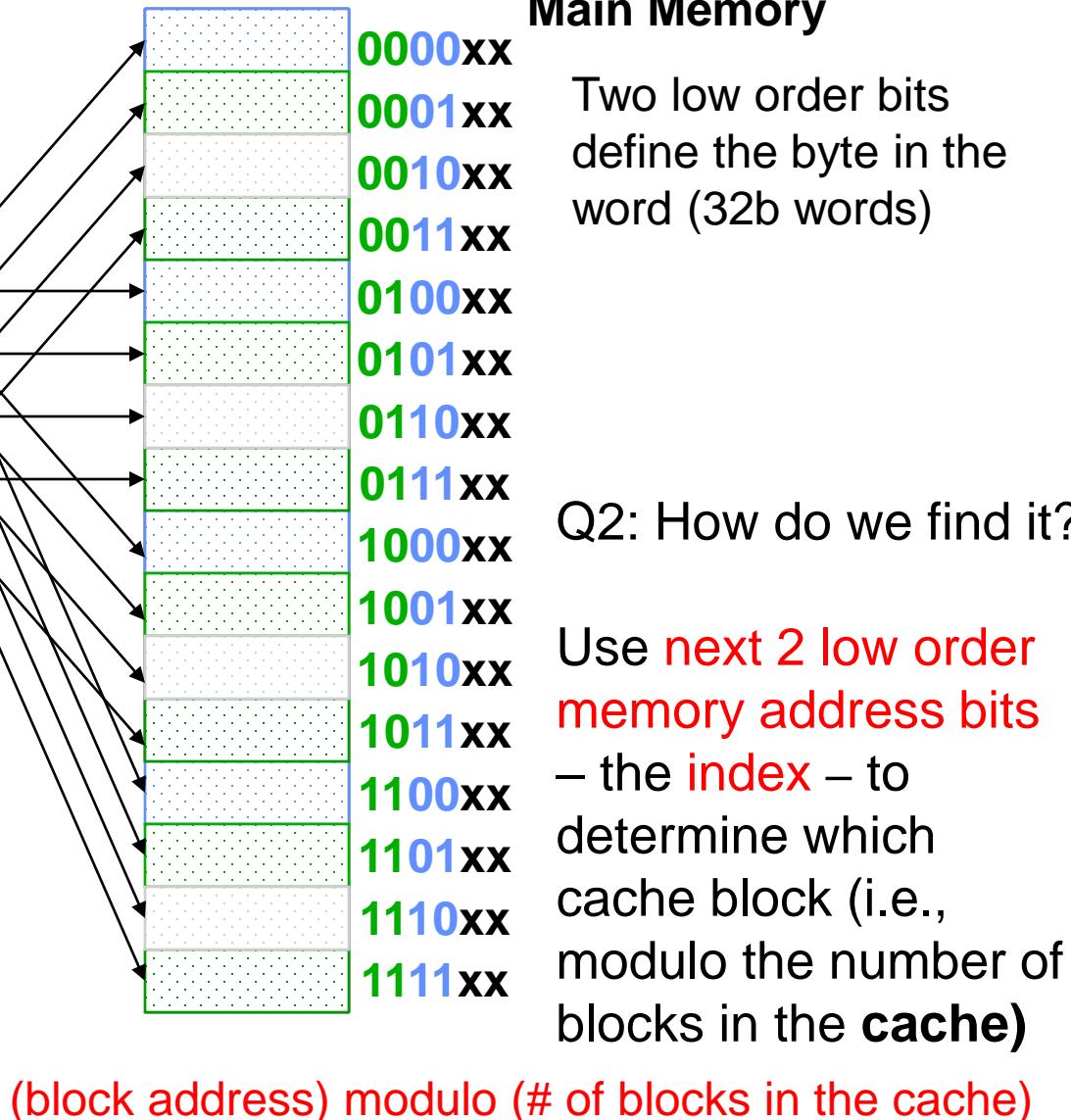
Cache

Index Valid Tag Data

00			0000xx
01			0001xx
10			0010xx
11			0011xx

Q1: Is it there?

Compare the cache tag to the **high order 2 memory address bits** to tell if the memory block is in the cache



Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

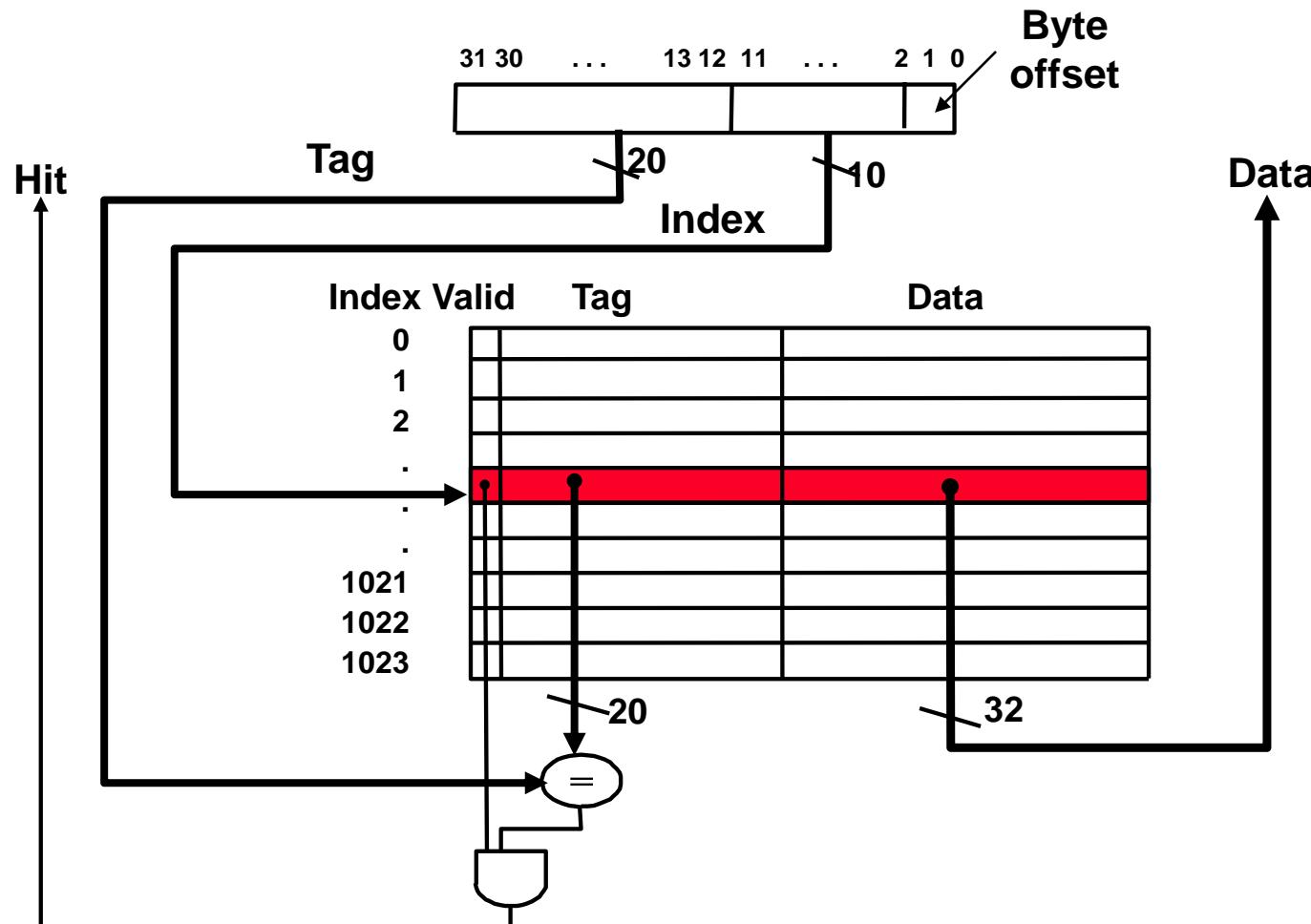
11

15

– 8 requests, 6 misses

MIPS Direct Mapped Cache Example

- One word/block, cache size = 1K words

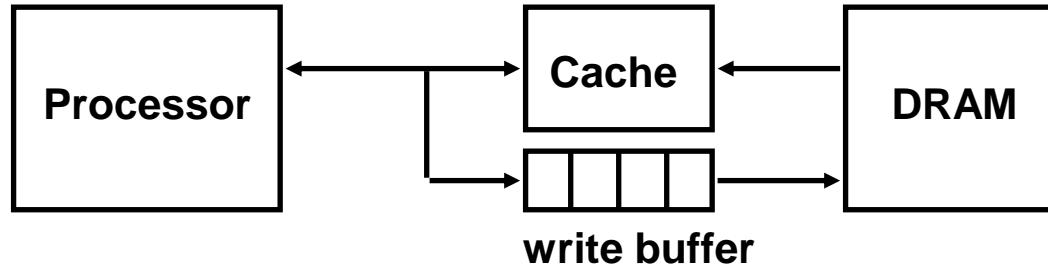


What kind of locality are we taking advantage of?

Handling Cache Hits

- Read hits (I\$ and D\$)
 - this is what we want!
- Write hits (D\$ only)
 - allow cache and memory to be **inconsistent**
 - write the data only into the cache block (**write-back** the cache contents to the next level in the memory hierarchy when that cache block is “evicted”)
 - need a **dirty** bit for each data cache block to tell if it needs to be written back to memory when it is evicted
 - require the cache and memory to be **consistent**
 - always write the data into both the cache block and the next level in the memory hierarchy (**write-through**) so don’t need a dirty bit
 - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a **write buffer**, so only have to stall if the write buffer is full

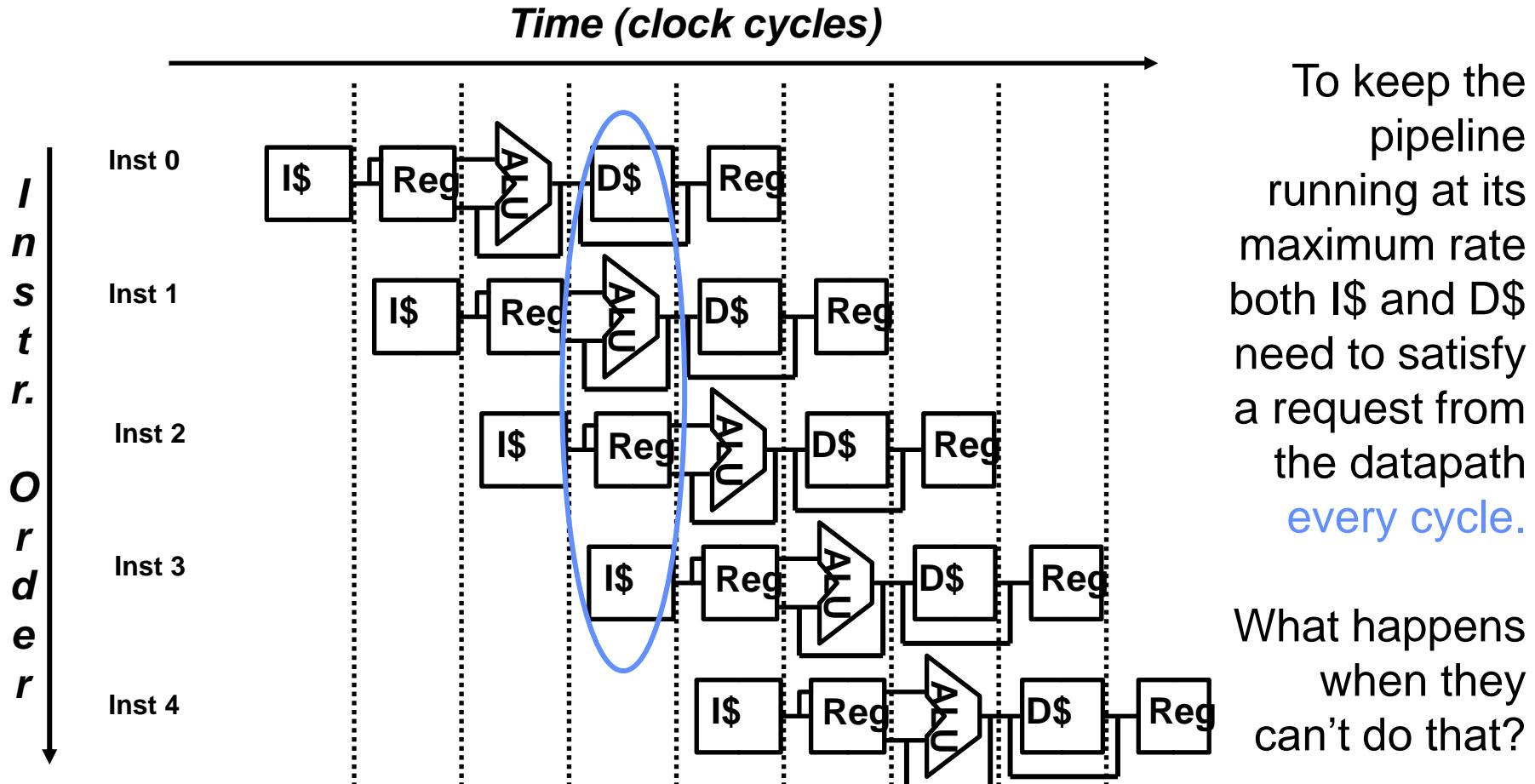
Write Buffer for Write-Through Caching



- Write buffer between the cache and main memory
 - Processor: writes data into the cache and the write buffer
 - Memory controller: writes contents of the write buffer to memory
- The write buffer is just a FIFO
 - Typical number of entries: 4
 - Works fine if **store frequency (w.r.t. time) << 1 / DRAM write cycle**
- Memory system designer's nightmare
 - When the **store frequency (w.r.t. time) → 1 / DRAM write cycle** leading to write buffer **saturation**
 - One solution is to use a write-back cache; another is to use an L2 cache

Review: Why Pipeline? For Throughput!

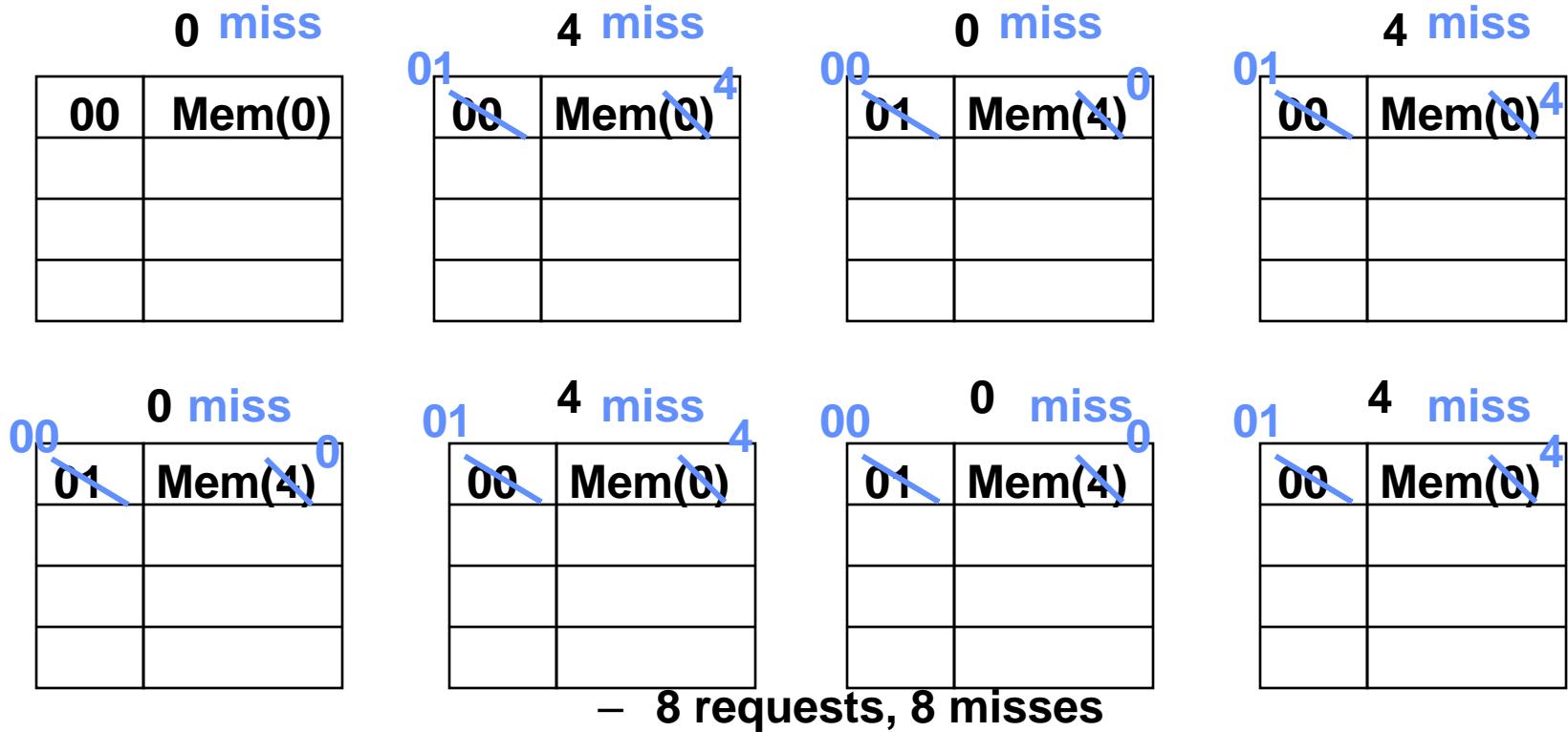
- To avoid a structural hazard need two caches on-chip: one for instructions (I\$) and one for data (D\$)



Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid



- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Sources of Cache Misses

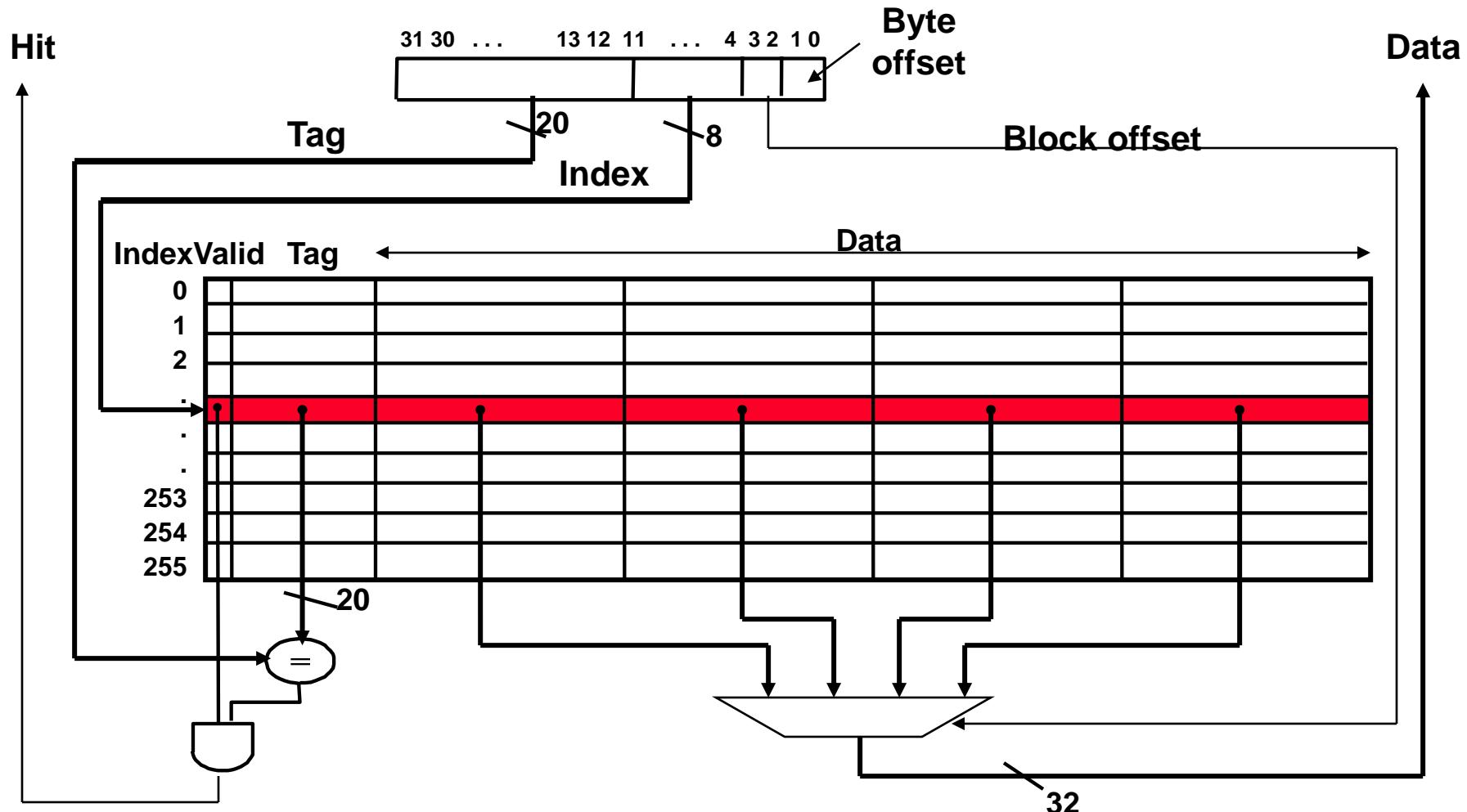
- **Compulsory** (cold start or process migration, first reference):
 - First access to a block, “cold” fact of life, not a whole lot you can do about it
 - If you are going to run “millions” of instruction, compulsory misses are insignificant
- **Conflict** (collision):
 - Multiple memory locations mapped to the same cache location
 - Solution 1: increase cache size
 - Solution 2: increase associativity
- **Capacity**:
 - Cache cannot contain all blocks accessed by the program
 - Solution: increase cache size

Handling Cache Misses

- Read misses (I\$ and D\$)
 - stall the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- Write misses (D\$ only)
 1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume
 - or (normally used in write-back caches)
 2. Write allocate – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall
 - or (normally used in write-through caches with a write buffer)
 3. No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full; must invalidate the cache block since it will be inconsistent (now holding stale data)

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?



Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

valid
0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

01	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

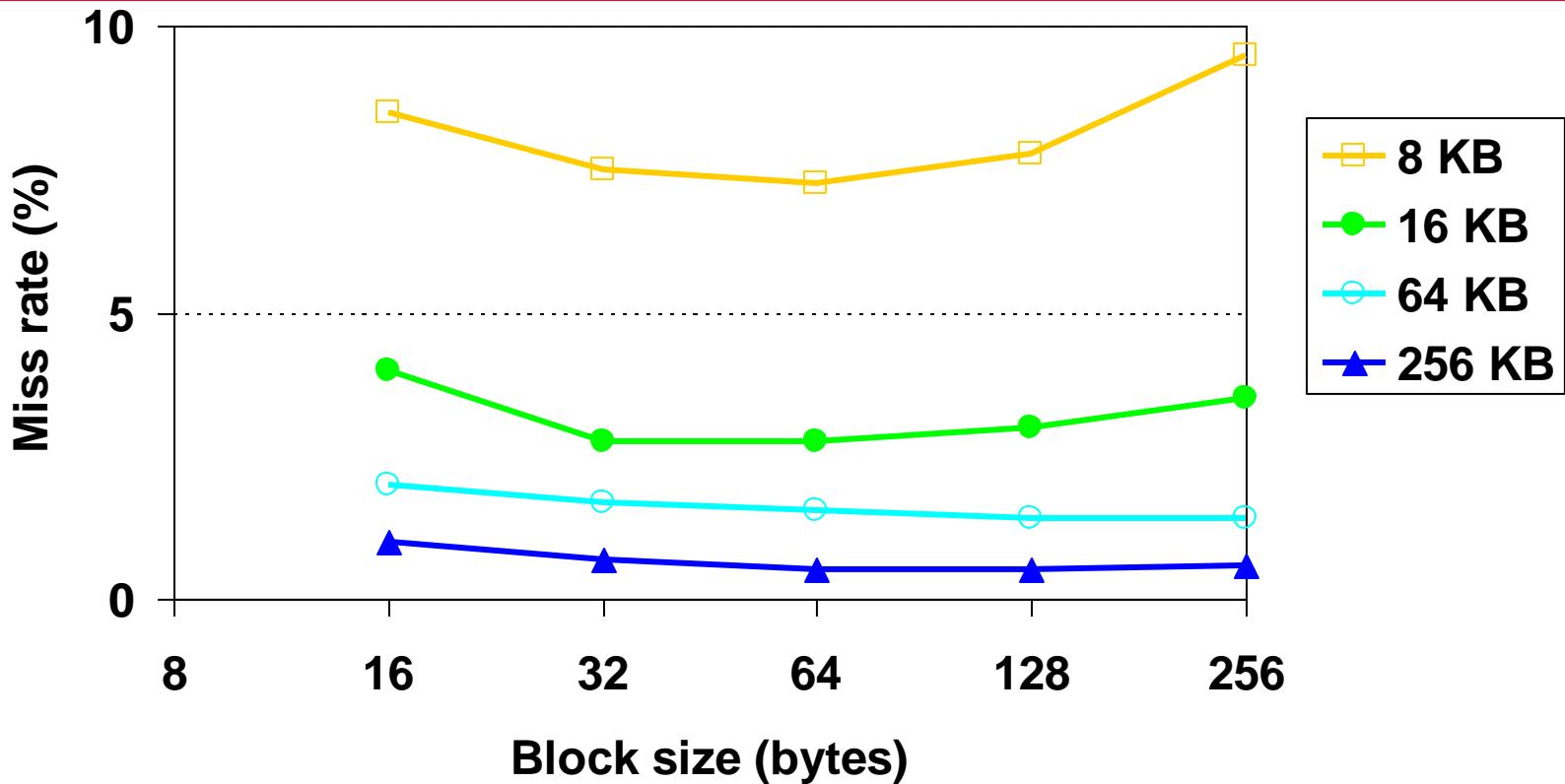
01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

11	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

– 8 requests, 4 misses

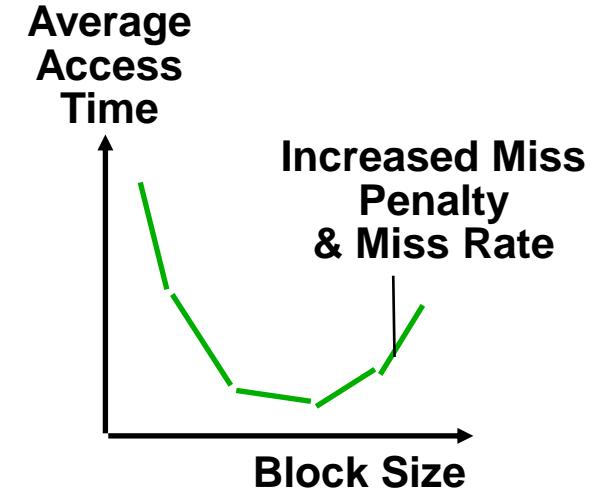
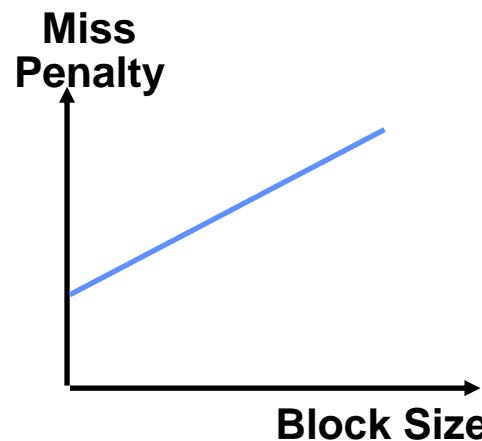
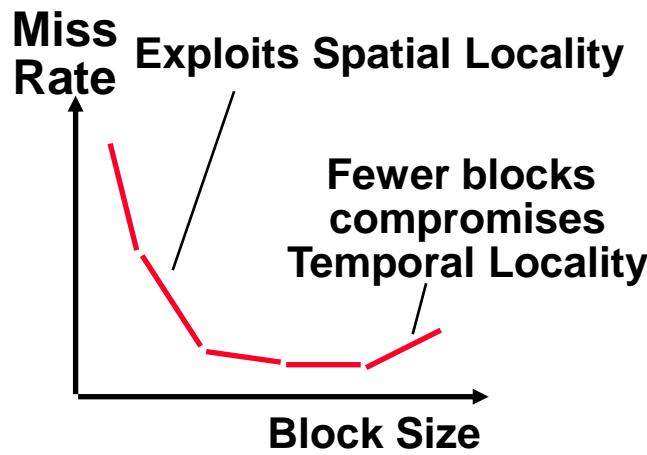
Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing **capacity misses**)

Block Size Tradeoff

- ❑ Larger block sizes take advantage of spatial locality but
 - If the block size is too big relative to the cache size, the miss rate will go up
 - Larger block size means larger miss penalty
 - Latency to first word in block + transfer time for remaining words



- ❑ In general, **Average Memory Access Time**
 $= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$

Multiword Block Considerations

- Read misses (I\$ and D\$)
 - Processed the same as for single word blocks – a miss returns the entire block from memory
 - Miss penalty grows as block size grows
 - **Early restart** – datapath resumes execution as soon as the requested word of the block is returned
 - **Requested word first** – requested word is transferred from the memory to the cache (and datapath) first
 - **Nonblocking cache** – allows the datapath to continue to access the cache while the cache is handling an earlier miss
- Write misses (D\$)
 - Can't use write allocate or will end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block), so must fetch the block from memory first and pay the stall time

Cache Summary

- The Principle of Locality:
 - Program likely to access a relatively small portion of the address space at any instant of time
 - **Temporal Locality**: Locality in Time
 - **Spatial Locality**: Locality in Space
- Three major categories of cache misses:
 - **Compulsory misses**: sad facts of life. Example: cold start misses
 - **Conflict misses**: increase cache size and/or associativity Nightmare Scenario: ping pong effect!
 - **Capacity misses**: increase cache size
- Cache design space
 - total size, block size, associativity (replacement policy)
 - write-hit policy (write-through, write-back)
 - write-miss policy (write allocate, write buffers)

ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 13: Memory Hierarchy

03/20/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Cache Summary

- The Principle of Locality:
 - Program likely to access a relatively small portion of the address space at any instant of time
 - **Temporal Locality**: Locality in Time
 - **Spatial Locality**: Locality in Space
- Three major categories of cache misses:
 - **Compulsory misses**: sad facts of life. Example: cold start misses
 - **Conflict misses**: increase cache size and/or associativity Nightmare Scenario: ping pong effect!
 - **Capacity misses**: increase cache size
- Cache design space
 - total size, block size, associativity (replacement policy)
 - write-hit policy (write-through, write-back)
 - write-miss policy (write allocate, write buffers)

Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\begin{aligned} \text{CPU time} &= \text{IC} \times \text{CPI} \times \text{CC} \\ &= \text{IC} \times (\underbrace{\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles}}_{\text{CPI}_{\text{stall}}} \times \text{CC}) \end{aligned}$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

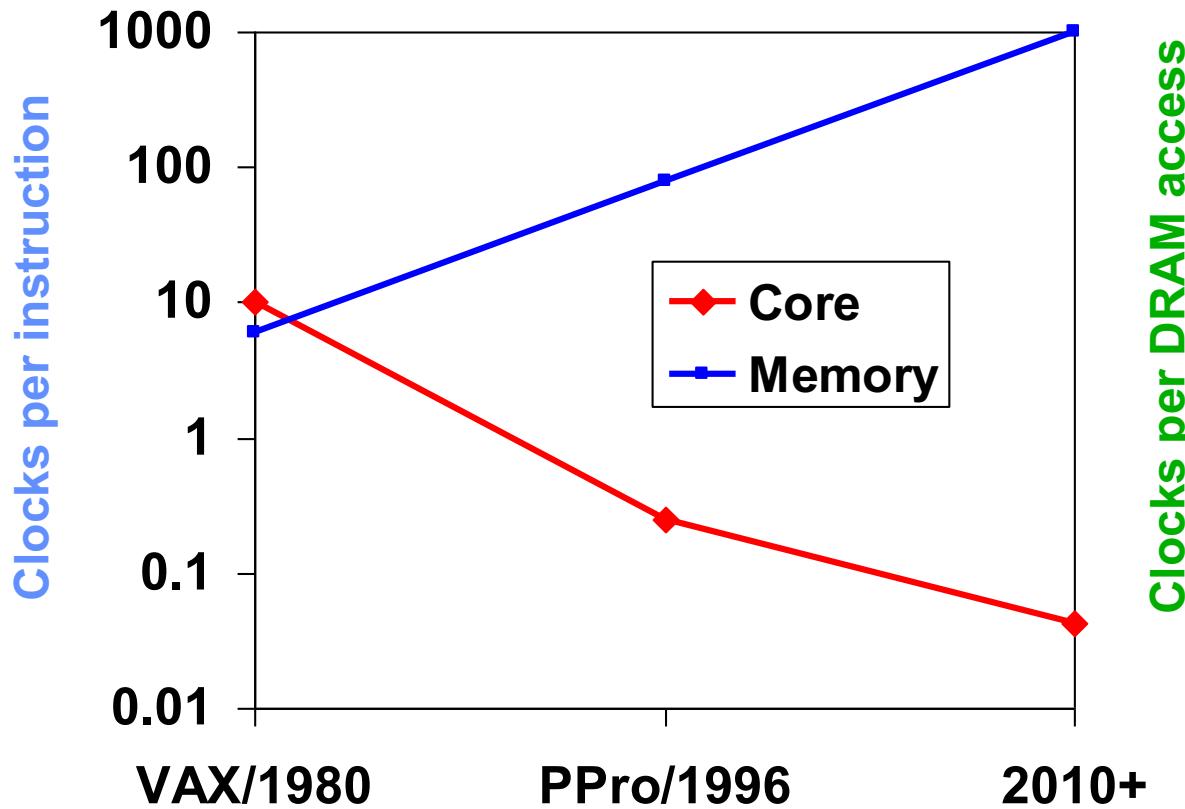
$$\begin{aligned} \text{Read-stall cycles} &= \text{reads/program} \times \text{read miss rate} \\ &\quad \times \text{read miss penalty} \end{aligned}$$

$$\begin{aligned} \text{Write-stall cycles} &= (\text{writes/program} \times \text{write miss rate} \\ &\quad \times \text{write miss penalty}) \\ &\quad + \text{write buffer stalls} \end{aligned}$$

- For write-through caches, we can simplify this to
- $$\text{Memory-stall cycles} = \text{miss rate} \times \text{miss penalty}$$

Review: The “Memory Wall”

- Logic vs DRAM speed gap continues to grow



Impacts of Cache Performance

- Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI)
 - The memory speed is unlikely to improve as fast as processor cycle time. When calculating CPI_{stall} , the cache miss penalty is measured in processor clock cycles needed to handle a miss
 - The lower the CPI_{ideal} , the more pronounced the impact of stalls
- A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% L\$ and 4% D\$ miss rates
$$\text{Memory-stall cycles} = 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$$
$$\text{So } CPI_{stalls} = 2 + 3.44 = 5.44$$
- What if the CPI_{ideal} is reduced to 1? 0.5? 0.25?
- What if the processor clock rate is doubled (doubling the miss penalty)?

Reducing Cache Miss Rates #1

1. Allow more flexible block placement

- In a **direct mapped cache** a memory block maps to exactly one cache block
- At the other extreme, could allow a memory block to be mapped to any cache block – **fully associative cache**
- A compromise is to divide the cache into **sets** each of which consists of n “ways” (**n -way set associative**). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)
(block address) modulo (# sets in the cache)

Set Associative Cache Example

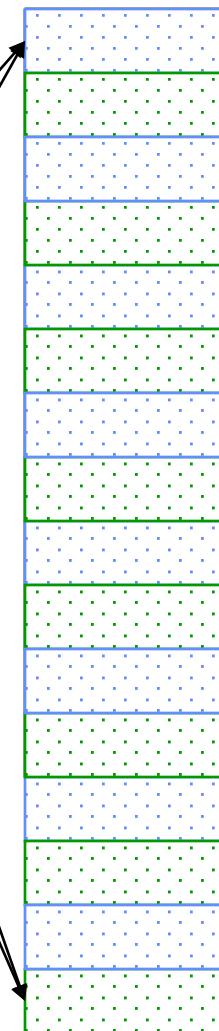
Cache

Way Set V Tag Data

	0	1	0	1
0				
1				

Q1: Is it there?

Compare *all* the cache **tags** in the set to the **high order 3 memory address bits** to tell if the memory block is in the cache



Main Memory

0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

Two low order bits define the byte in the word (32-b words)
One word blocks

Q2: How do we find it?

Use **next 1 low order memory address bit** to determine which cache set (i.e., modulo the number of sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

4 hit

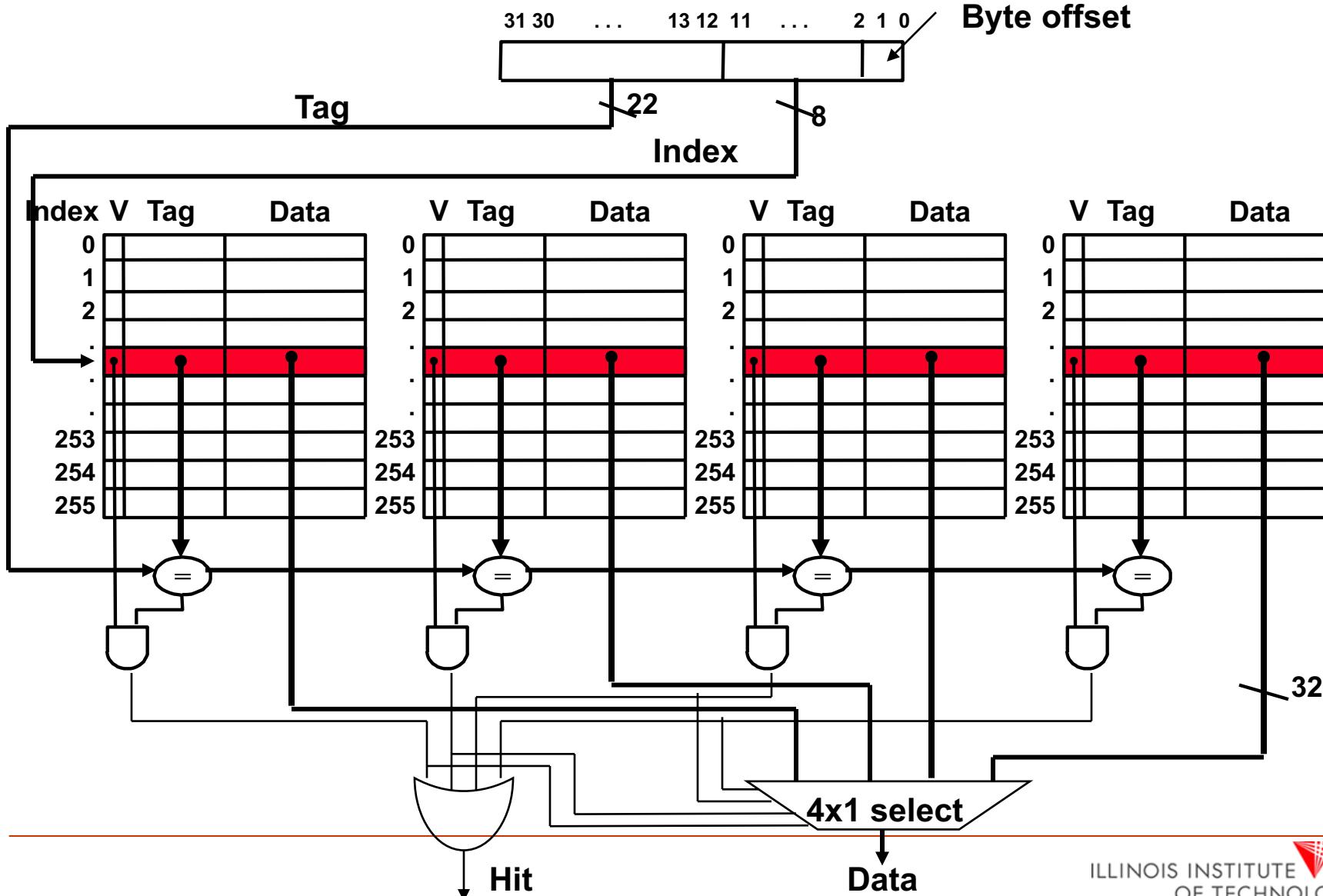
000	Mem(0)
010	Mem(4)

– 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to **conflict** misses since now two memory locations that map into the same cache set can co-exist!

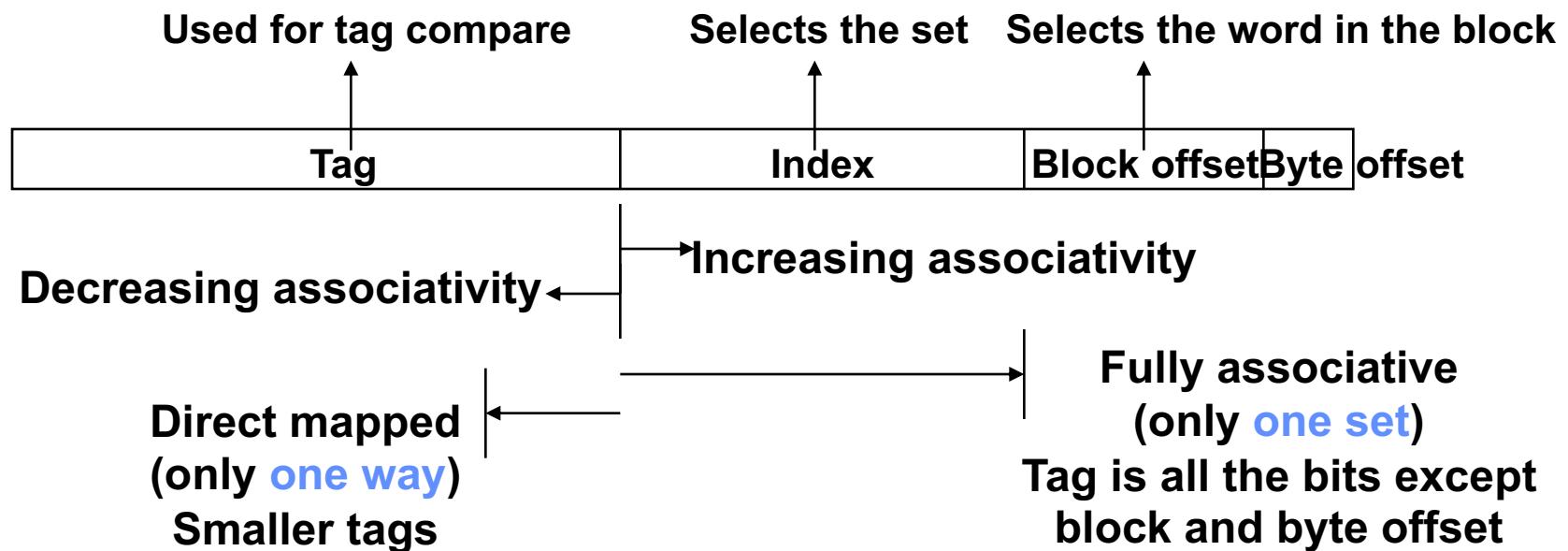
Four-Way Set Associative Cache

- $2^8 = 256$ sets each with four ways (each with one block)



Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

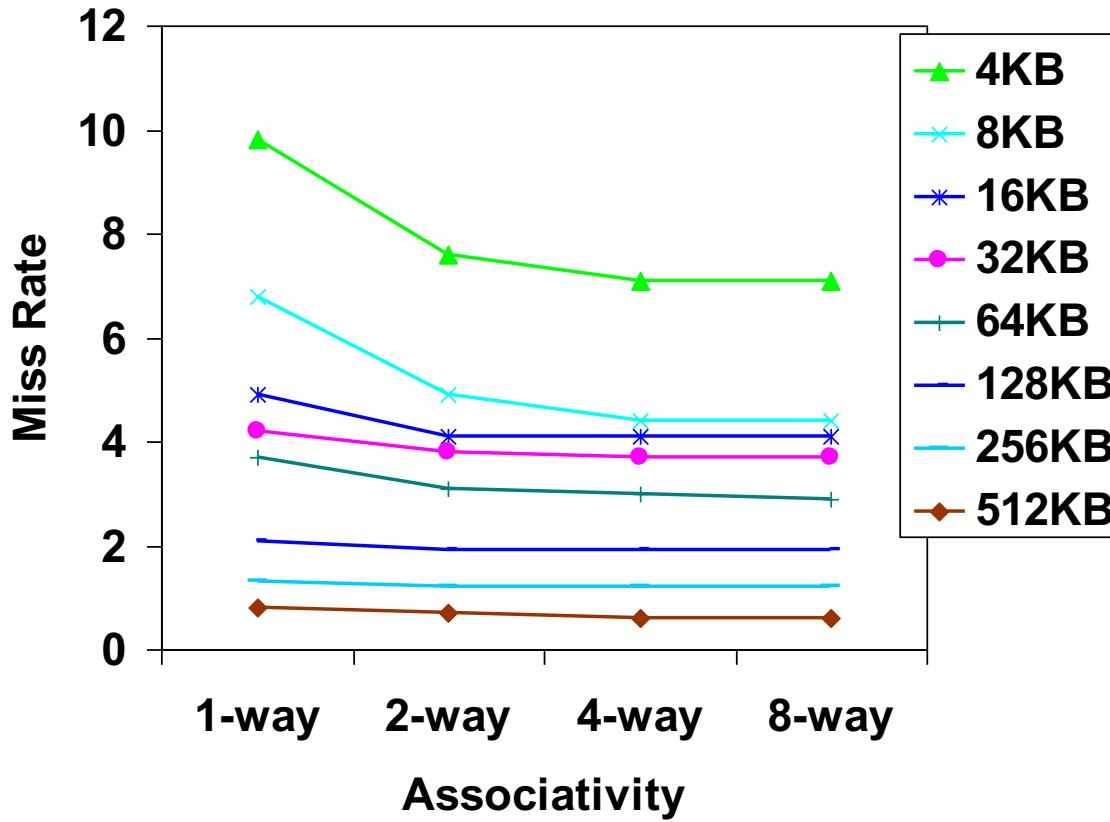


Costs of Set Associative Caches

- When a miss occurs, which way's block do we pick for replacement?
 - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
 - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
 - For 2-way set associative, takes **one bit per set** → set the bit when a block is referenced (and reset the other way's bit)
- N-way set associative cache costs
 - N comparators (delay and area)
 - MUX delay (set selection) before data is available
 - Data available **after** set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available **before** the Hit/Miss decision
 - So it's not possible to just assume a hit and continue and recover later if it was a miss

Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Reducing Cache Miss Rates #2

2. Use multiple levels of caches

- With advancing technology have more than enough room on the die for bigger L1 caches or for a second level of caches – normally a **unified** L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache
- For our example, CPI_{ideal} of 2, 100 cycle miss penalty (to main memory), 36% load/stores, a 2% (4%) L1I\$ (D\$) miss rate, add a UL2\$ that has a 25 cycle miss penalty and a 0.5% miss rate

$$CPI_{stalls} = 2 + .02 \times 25 + .36 \times .04 \times 25 + .005 \times 100 + .36 \times .005 \times 100 = 3.54$$

(as compared to 5.44 with no L2\$)

Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
 - Primary cache should focus on **minimizing hit time** in support of a shorter clock cycle
 - Smaller with smaller block sizes
 - Secondary cache(s) should focus on **reducing miss rate** to reduce the penalty of long main memory access times
 - Larger with larger block sizes
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
 - The L2\$ hit time determines L1\$'s miss penalty
 - L2\$ local miss rate \gg than the global miss rate

Key Cache Design Parameters

	L1 typical	L2 typical
Total size (blocks)	250 to 2000	4000 to 250,000
Total size (KB)	16 to 64	500 to 8000
Block size (B)	32 to 64	32 to 128
Miss penalty (clocks)	10 to 25	100 to 1000
Miss rates (global for L2)	2% to 5%	0.1% to 2%

Two Machines' Cache Parameters

	Intel P4	AMD Opteron
L1 organization	Split I\$ and D\$	Split I\$ and D\$
L1 cache size	8KB for D\$, 96KB for trace cache (~I\$)	64KB for each of I\$ and D\$
L1 block size	64 bytes	64 bytes
L1 associativity	4-way set assoc.	2-way set assoc.
L1 replacement	~ LRU	LRU
L1 write policy	write-through	write-back
L2 organization	Unified	Unified
L2 cache size	512KB	1024KB (1MB)
L2 block size	128 bytes	64 bytes
L2 associativity	8-way set assoc.	16-way set assoc.
L2 replacement	~LRU	~LRU
L2 write policy	write-back	write-back

Intel I7- Nehalem

Intel Nehalem

A Research Report

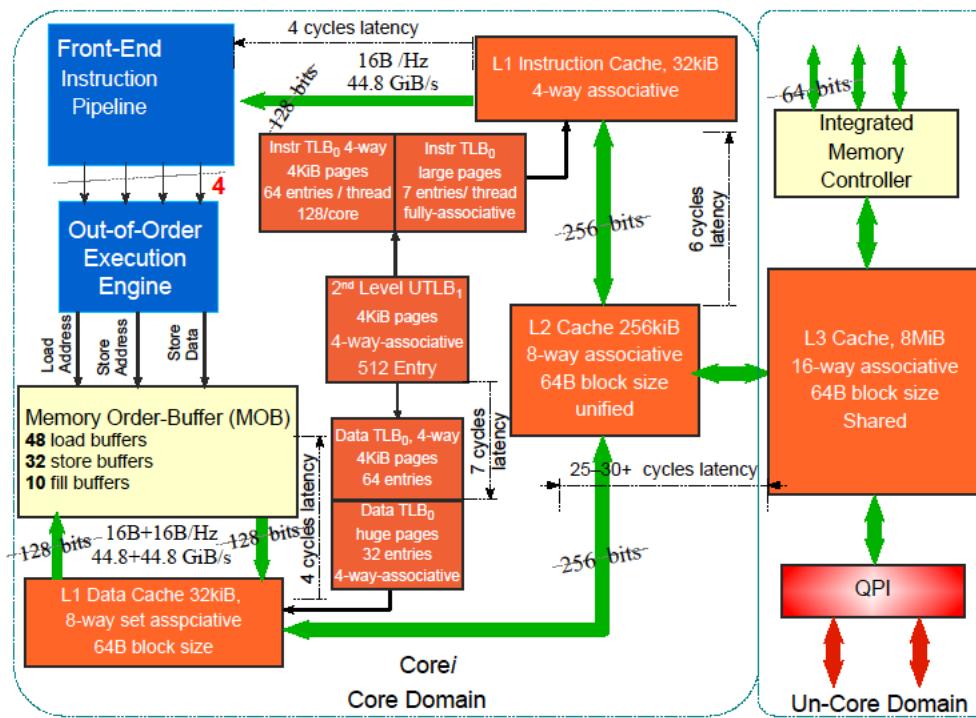


Figure 10: Overview of Cache Memory Hierarchy and Data Flow Paths to and from Nehalem's Core.

4 Questions for the Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
(Block placement)
- Q2: How is a block found if it is in the upper level?
(Block identification)
- Q3: Which block should be replaced on a miss?
(Block replacement)
- Q4: What happens on a write?
(Write strategy)

Q1&Q2: Where can a block be placed/found?

	# of sets	Blocks per set
Direct mapped	# of blocks in cache	1
Set associative	(# of blocks in cache)/associativity	Associativity (typically 2 to 16)
Fully associative	1	# of blocks in cache

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all blocks tags	# of blocks

Q3: Which block should be replaced on a miss?

- Easy for direct mapped – only one choice
- Set associative or fully associative
 - Random
 - LRU (Least Recently Used)
- For a 2-way set associative cache, random replacement has a miss rate about 1.1 times higher than LRU.
- LRU is too costly to implement for high levels of associativity ($> 4\text{-way}$) since tracking the usage information is costly

Q4: What happens on a write?

- Write-through – The information is written to both the block in the cache and to the block in the next lower level of the memory hierarchy
 - Write-through is always combined with a write buffer so write waits to lower level memory can be eliminated (as long as the write buffer doesn't fill)
- Write-back – The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
 - Need a dirty bit to keep track of whether the block is clean or dirty
- Pros and cons of each?
 - Write-through: read misses don't result in writes (so are simpler and cheaper)
 - Write-back: repeated writes require only one write to lower level

Improving Cache Performance

0. Reduce the time to hit in the cache

- smaller cache
- direct mapped cache
- smaller blocks
- for writes
 - no write allocate – no “hit” on cache, just write to write buffer
 - write allocate – to avoid two cycles (first check for hit, then write)
pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate

- bigger cache
- more flexible placement (increase associativity)
- larger blocks (16 to 64 bytes typical)
- victim cache – small buffer holding most recently discarded blocks

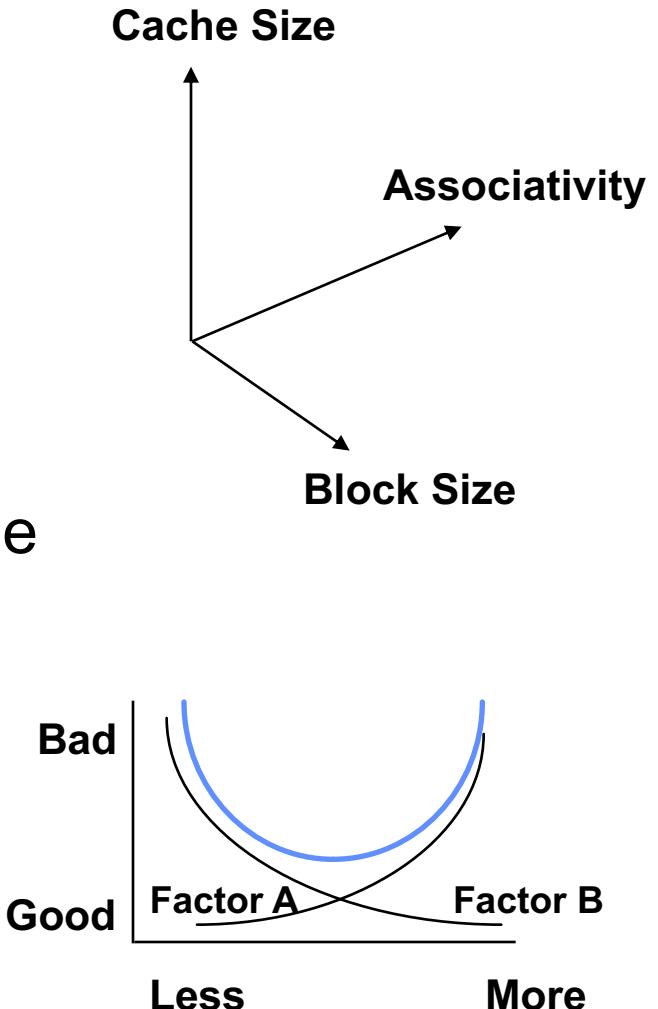
Improving Cache Performance

2. Reduce the miss penalty

- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss – may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
 - wider buses
 - memory interleaving, page mode DRAMs

Summary: The Cache Design Space

- Several interacting dimensions
 - cache size
 - block size
 - associativity
 - replacement policy
 - write-through vs write-back
 - write allocation
- The optimal choice is a compromise
 - depends on access characteristics
 - workload
 - use (I-cache, D-cache, TLB)
 - depends on technology / cost
- Simplicity often wins



ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 14: **Virtual Memory**

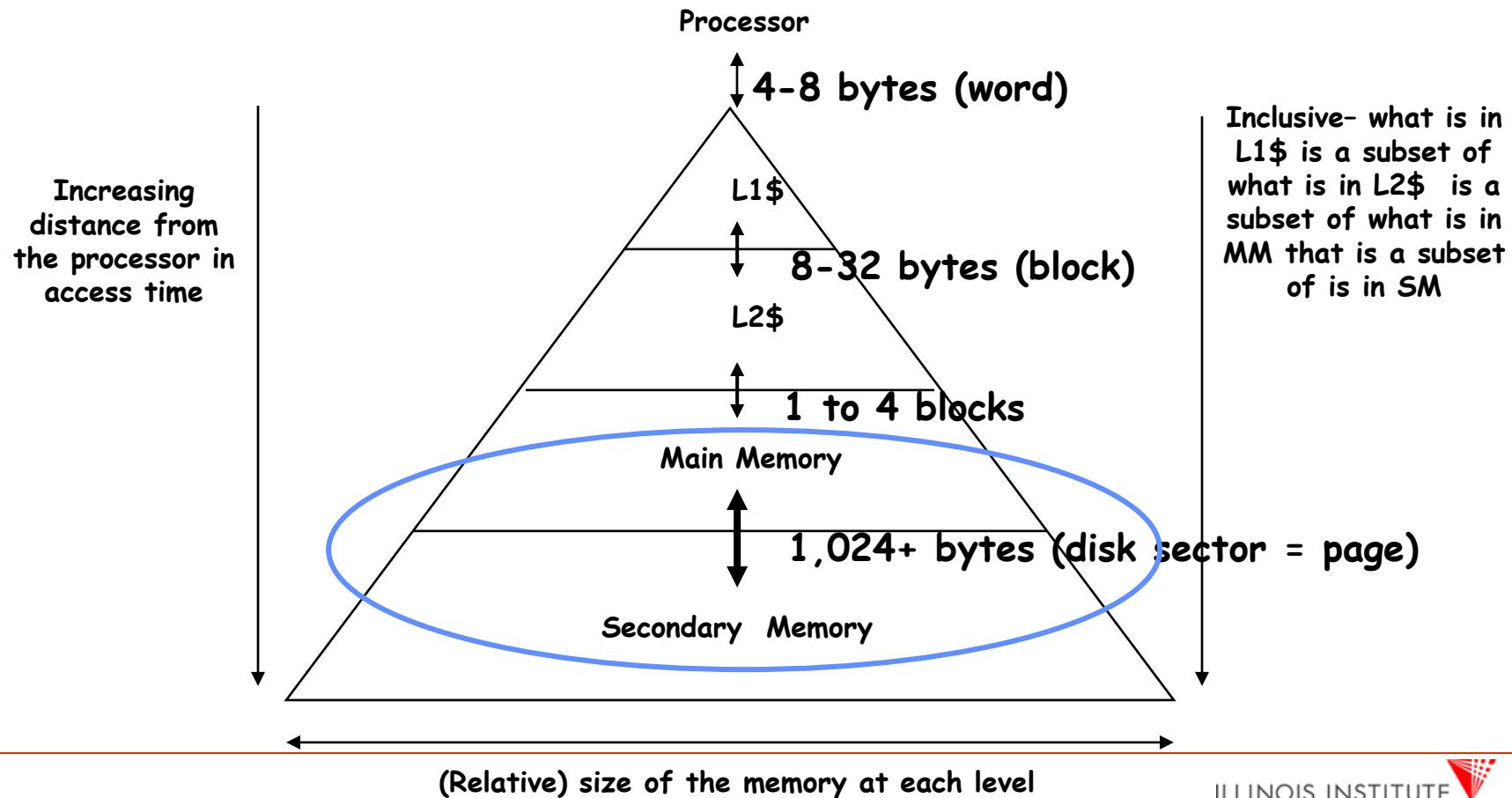
03/27/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Review: The Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



Virtual Memory

- Use main memory as a “cache” for secondary memory
 - Allows efficient and safe sharing of memory among multiple programs
 - Provides the ability to easily run programs larger than the size of physical memory
 - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
- What makes it work? – again the Principle of Locality
 - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
 - During run-time each **virtual** address must be translated to a **physical** address (an address in main memory)

Virtual Memory

- Protection via virtual memory
 - Keeps processes in their own memory space
- Role of architecture:
 - Provide user mode and supervisor mode
 - Protect certain aspects of CPU state
 - Provide mechanisms for switching between user mode and supervisor mode
 - Provide mechanisms to limit memory accesses
 - Provide TLB to translate addresses

Virtual Machines

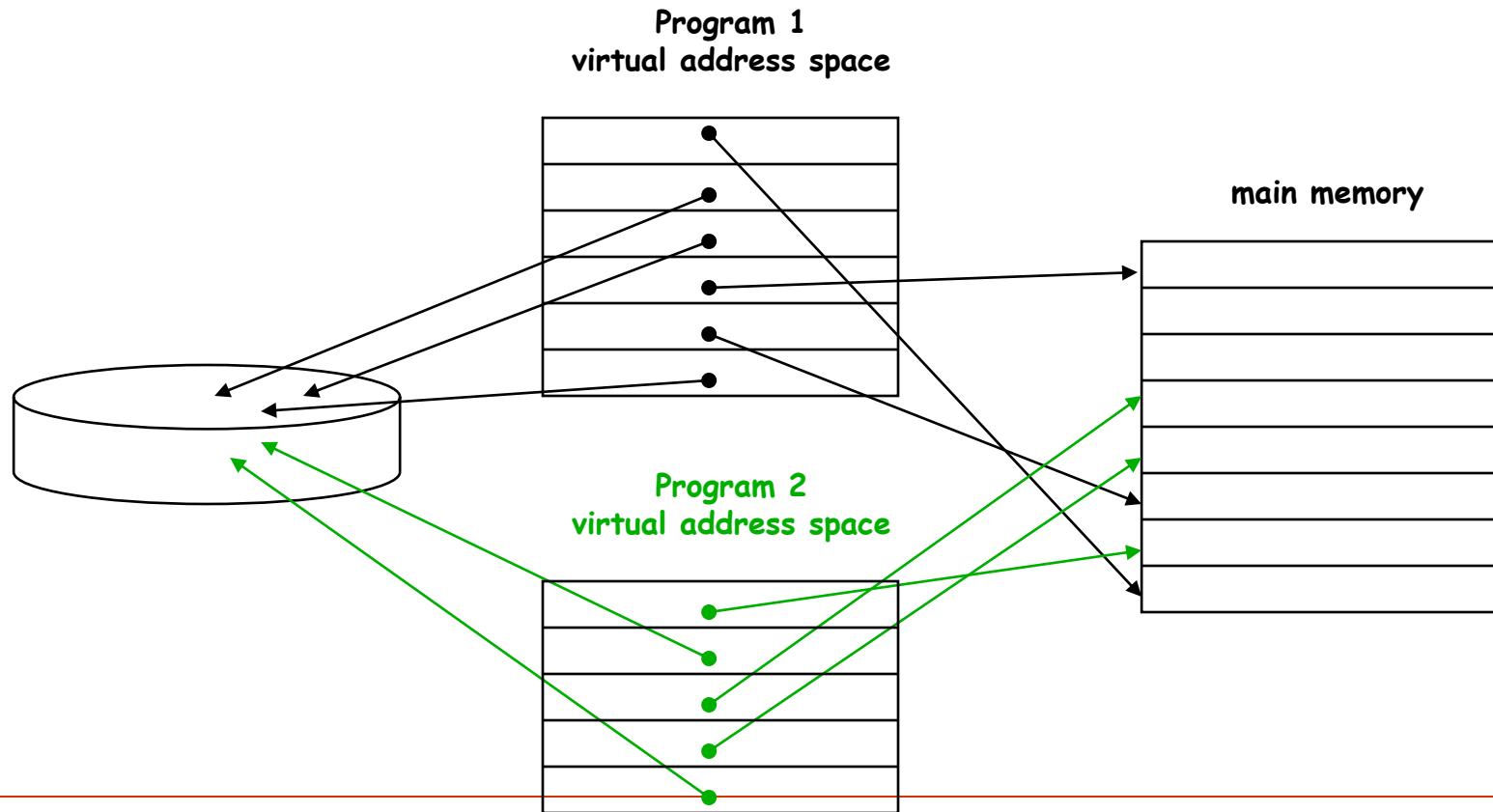
- Supports isolation and security
- Sharing a computer among many unrelated users
- Enabled by raw speed of processors, making the overhead more acceptable
- Allows different ISAs and operating systems to be presented to user programs
 - “System Virtual Machines”
 - SVM software is called “virtual machine monitor” or “hypervisor”
 - Individual virtual machines run under the monitor are called “guest VMs”

Impact of VMs on Virtual Memory

- Each guest OS maintains its own set of page tables
 - VMM adds a level of memory between physical and virtual memory called “real memory”
 - VMM maintains shadow page table that maps guest virtual addresses to physical addresses
 - Requires VMM to detect guest’s changes to its own page table
 - Occurs naturally if accessing the page table pointer is a privileged operation

Two Programs Sharing Physical Memory

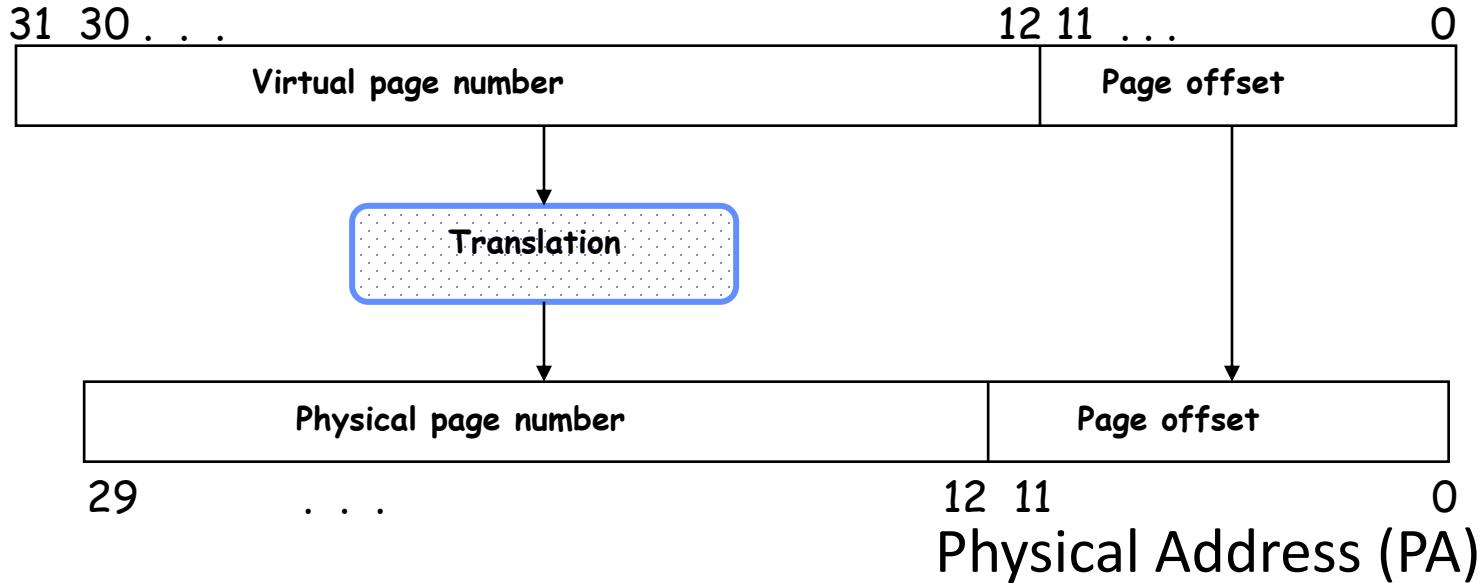
- A program's address space is divided into **pages** (all one fixed size) or segments (variable sizes)
 - The starting location of each page (either in main memory or in secondary memory) is contained in the program's **page table**



Address Translation

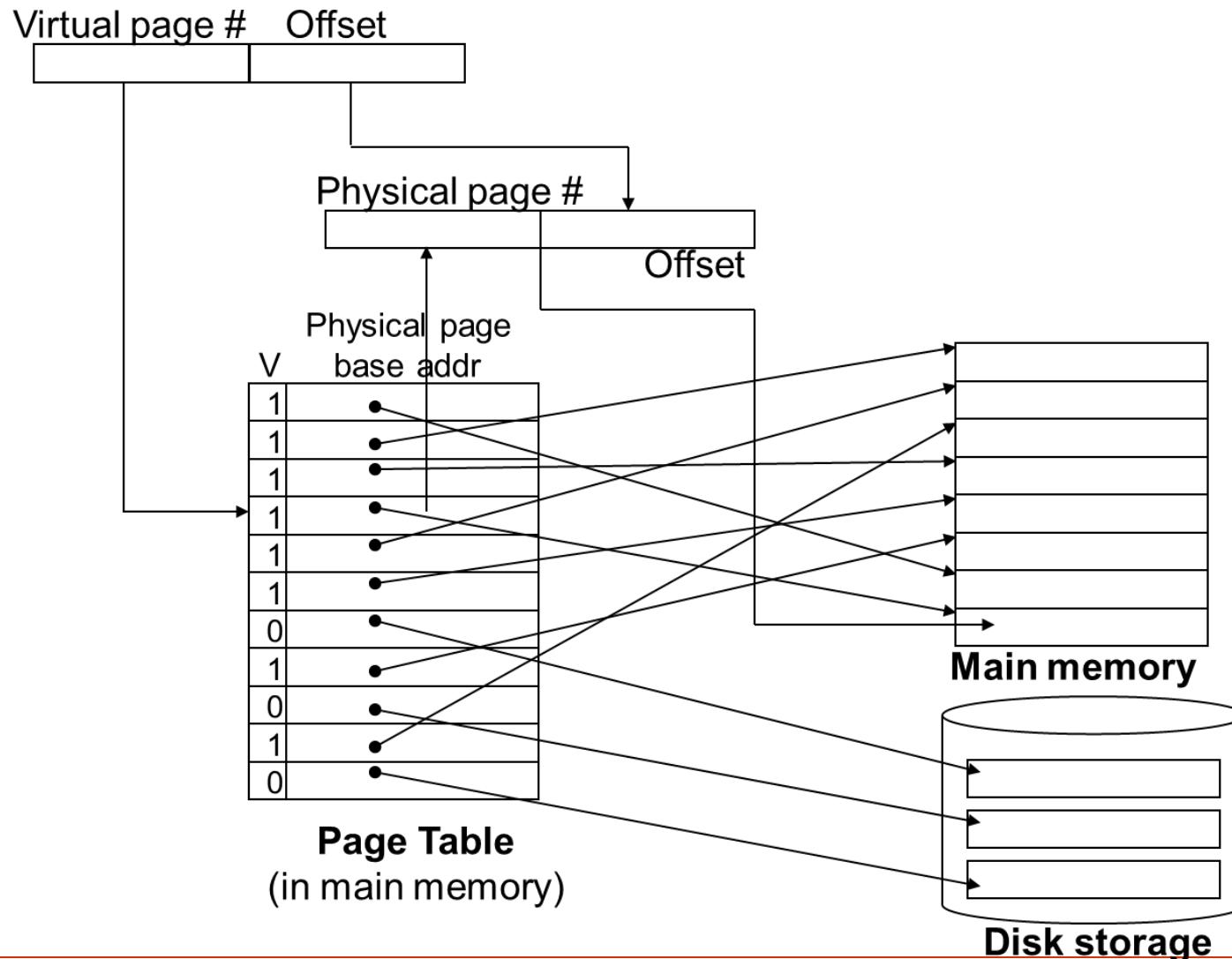
- A **virtual address** is translated to a **physical address** by a combination of hardware and software

Virtual Address (VA)



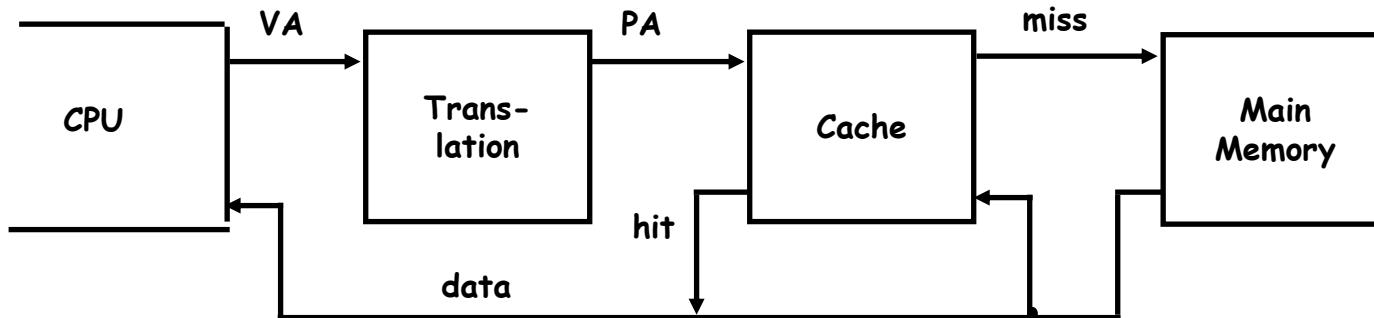
- So each memory request *first* requires an address **translation** from the virtual space to the physical space
 - A virtual memory miss (i.e., when the page is not in physical memory) is called a **page fault**

Address Translation Mechanisms



Virtual Addressing with a Cache

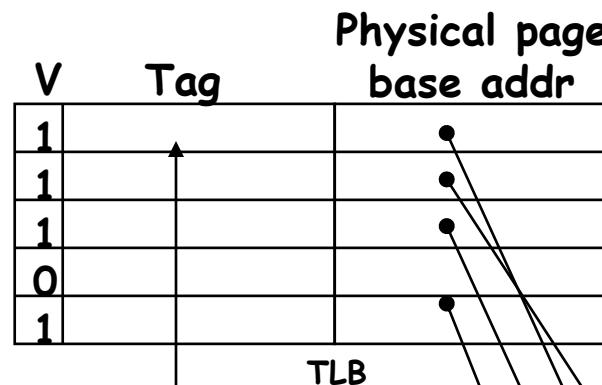
- Thus it takes an *extra* memory access to translate a VA to a PA



- This makes memory (cache) accesses **very expensive** (if every access was really *two* accesses)
- The hardware fix is to use a Translation Lookaside Buffer (TLB)
 - a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

Making Address Translation Fast

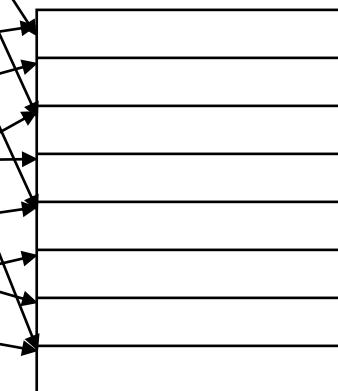
Virtual page #



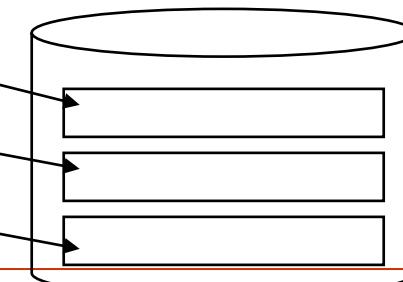
Physical page
V base addr

1	•
1	•
1	•
1	•
1	•
1	•
0	•
1	•
0	•
1	•
0	•

Page Table
(in physical memory)



Main memory



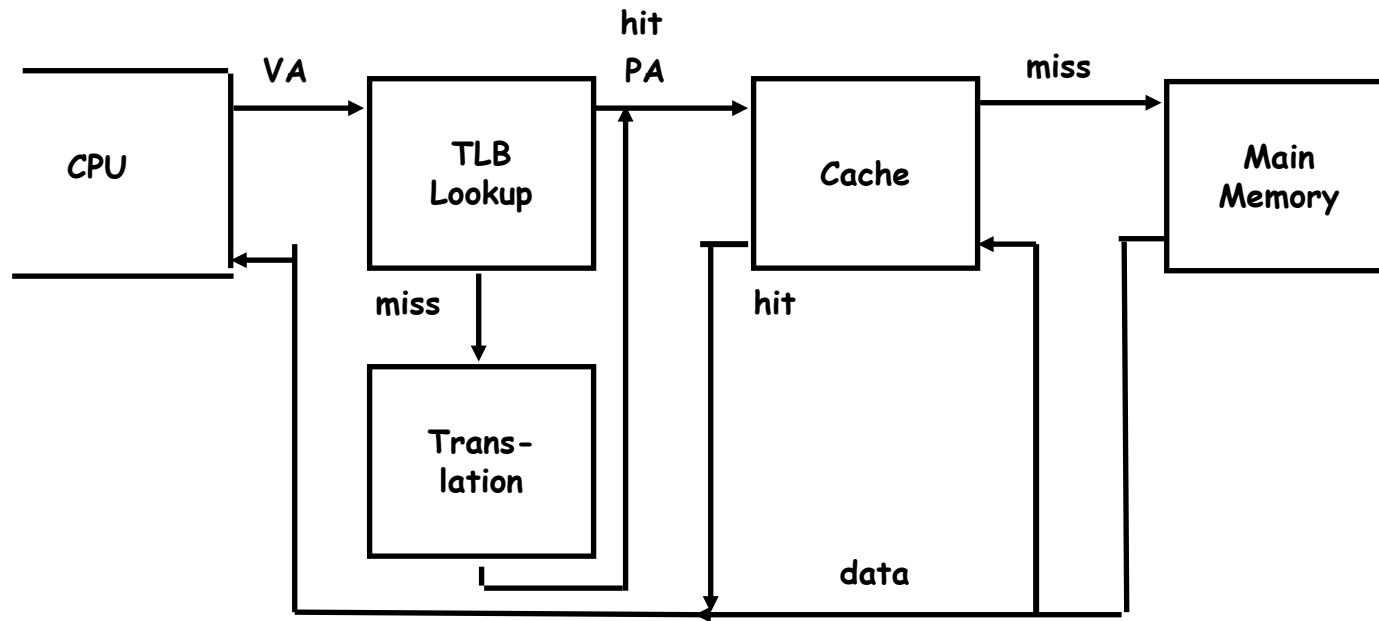
Translation Lookaside Buffers (TLBs)

- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

V	Virtual Page #	Physical Page #	Dirty	Ref	Access

- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
 - TLBs are typically not more than 128 to 256 entries even on high end machines

A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
 - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
 - Takes 10's of cycles to find and load the translation info into the TLB
 - If the page is not in main memory, then it's a true page fault
 - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

Some Virtual Memory Design Parameters

	Paged VM	TLBs
Total size	16,000 to 250,000 words	16 to 512 entries
Total size (KB)	250,000 to 1,000,000,000	0.25 to 16
Block size (B)	4000 to 64,000	4 to 32
Miss penalty (clocks)	10,000,000 to 100,000,000	10 to 1000
Miss rates	0.00001% to 0.0001%	0.01% to 2%

Two Machines' Cache Parameters

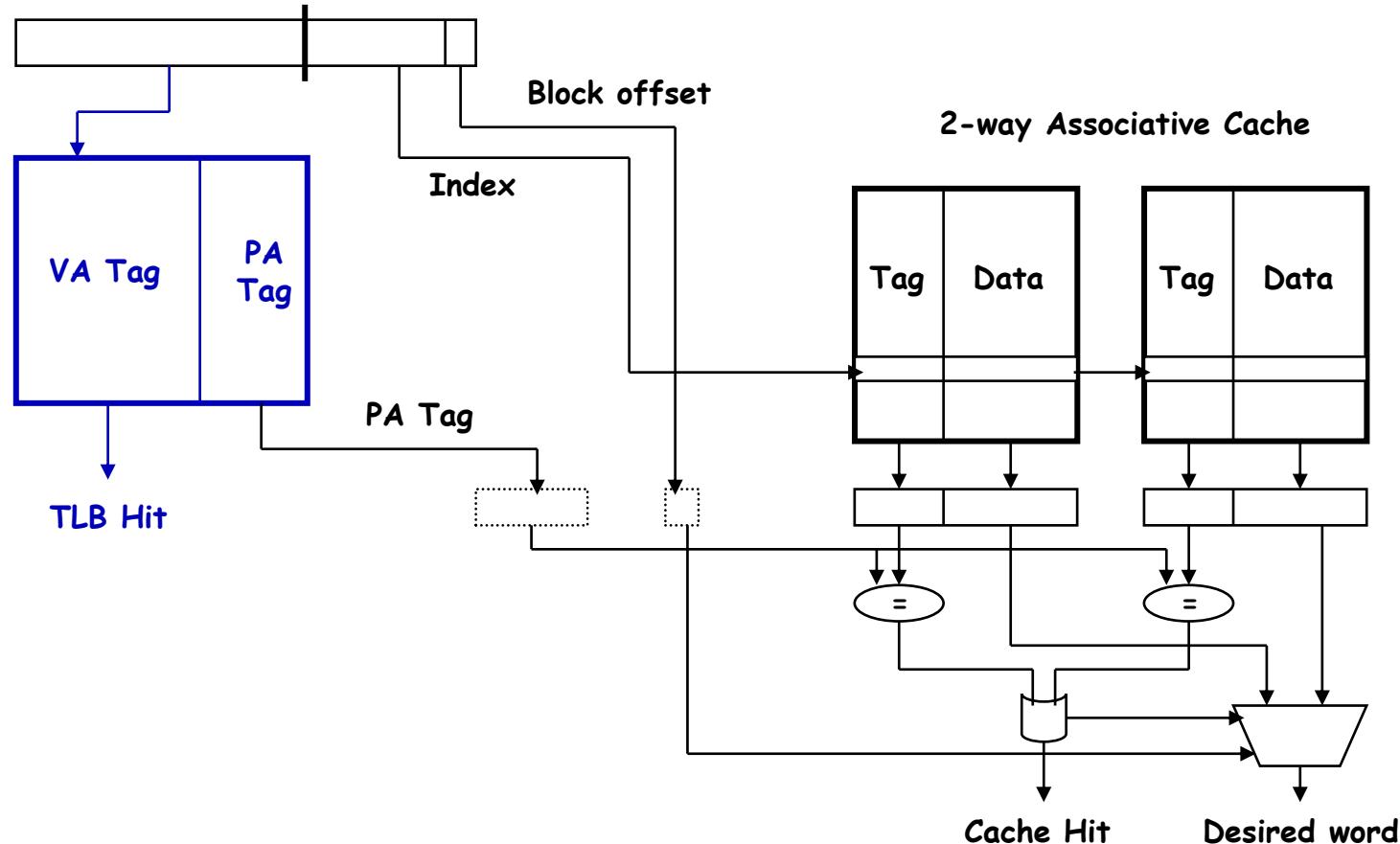
	Intel P4	AMD Opteron
TLB organization	1 TLB for instructions and 1TLB for data Both 4-way set associative Both use ~LRU replacement Both have 128 entries TLB misses handled in hardware	2 TLBs for instructions and 2 TLBs for data Both L1 TLBs fully associative with ~LRU replacement Both L2 TLBs are 4-way set associative with round-robin LRU Both L1 TLBs have 40 entries Both L2 TLBs have 512 entries TBL misses handled in hardware

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

Reducing Translation Time

- Can **overlap** the cache access with the TLB access
 - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache
 - Virtually indexed, physically tagged cache



The Hardware/Software Boundary

- What parts of the virtual to physical address translation is done by or assisted by the hardware?
 - Translation Lookaside Buffer (TLB) that caches the recent translations
 - TLB access time is part of the cache hit time
 - May allot an extra stage in the pipeline for TLB access
 - Page table storage, fault detection and updating
 - Page faults result in interrupts (precise) that are then handled by the OS
 - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., \sim LRU) in the Page Tables
 - Disk placement
 - Bootstrap (e.g., out of disk sector 0) so the system can service a limited number of page faults before the OS is even loaded

Summary

- The Principle of Locality:
 - Program likely to access a relatively small portion of the address space at any instant of time.
 - **Temporal Locality**: Locality in Time
 - **Spatial Locality**: Locality in Space
- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
 1. Where can block be placed?
 2. How is block found?
 3. What block is replaced on miss?
 4. How are writes handled?
- Page tables map virtual address to physical address
 - TLBs are important for fast translation

Hypothetical Memory Hierarchy - 1

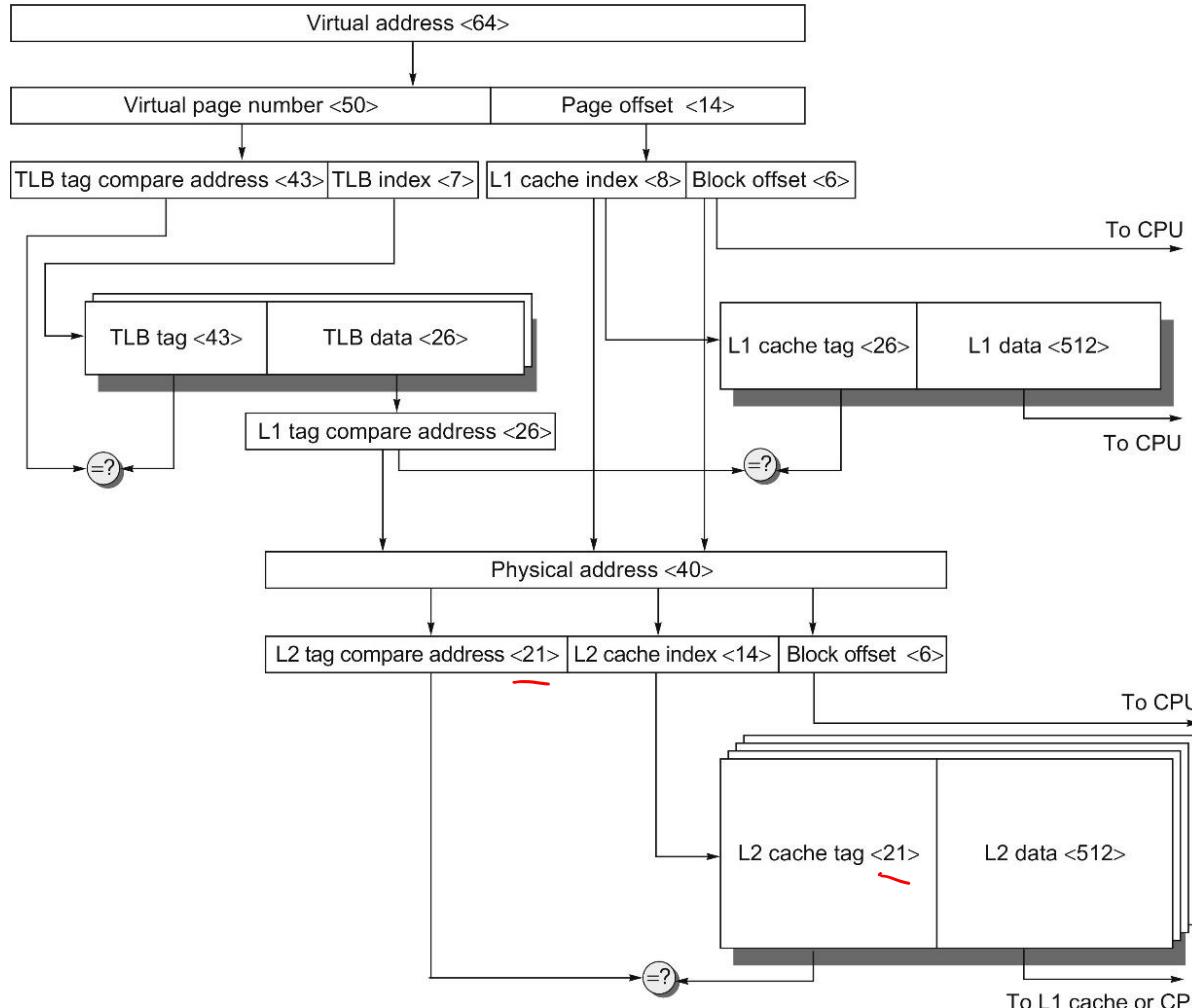


Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KiB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KiB, and the L2 cache is a four-way set associative with a total of 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Hypothetical Memory Hierarchy -2

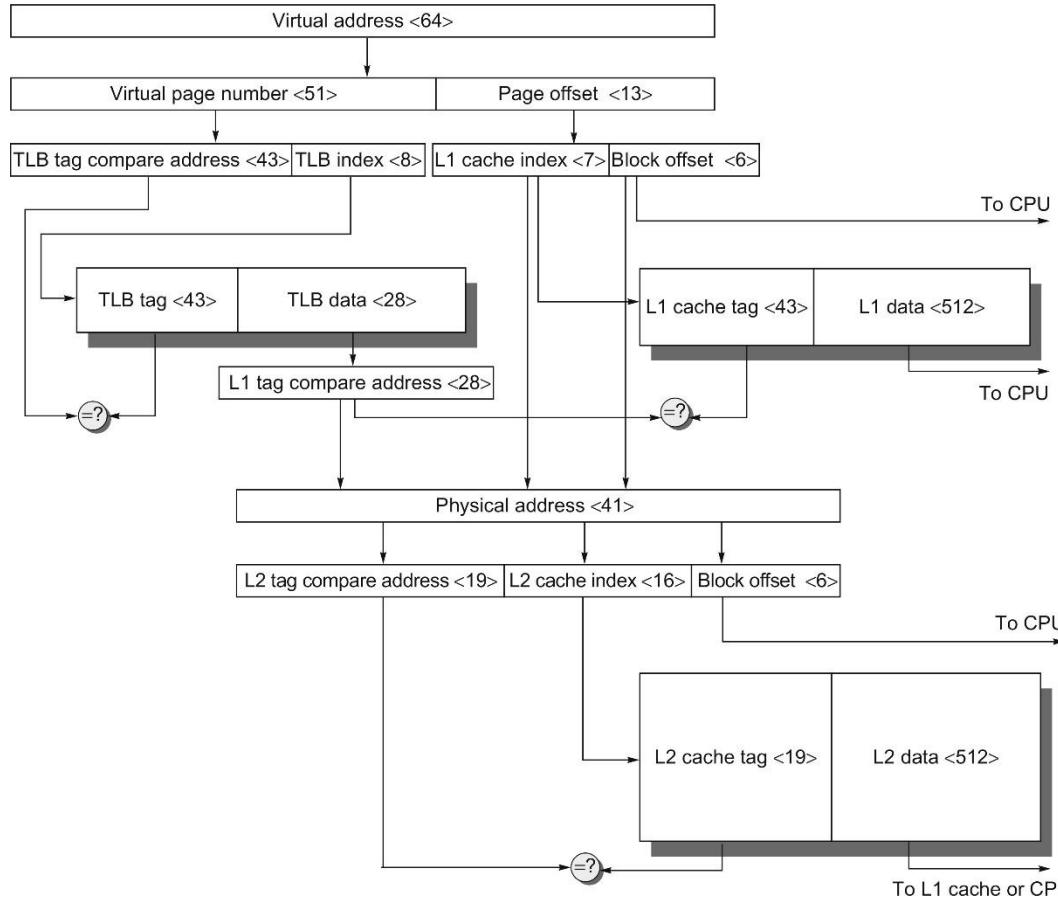


Figure B.25 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KiB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KiB, and the L2 cache is a direct-mapped 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

Arm Cortex A53

Structure	Size	Organization	Typical miss penalty (clock cycles)
Instruction MicroTLB	10 entries	Fully associative	2
Data MicroTLB	10 entries	Fully associative	2
L2 Unified TLB	512 entries	4-way set associative	20
L1 Instruction cache	8–64 KiB	2-way set associative; 64-byte block	13
L1 Data cache	8–64 KiB	2-way set associative; 64-byte block	13
L2 Unified cache	128 KiB to 2 MiB	16-way set associative; LRU	124

Figure 2.19 The memory hierarchy of the Cortex A53 includes multilevel TLBs and caches. A page map cache keeps track of the location of a physical page for a set of virtual pages; it reduces the L2 TLB miss penalty. The L1 caches are virtually indexed and physically tagged; both the L1 D cache and L2 use a write-back policy defaulting to allocate on write. Replacement policy is LRU approximation in all the caches. Miss penalties to L2 are higher if both a MicroTLB and L1 miss occur. The L2 to main memory bus is 64–128 bits wide, and the miss penalty is larger for the narrow bus.

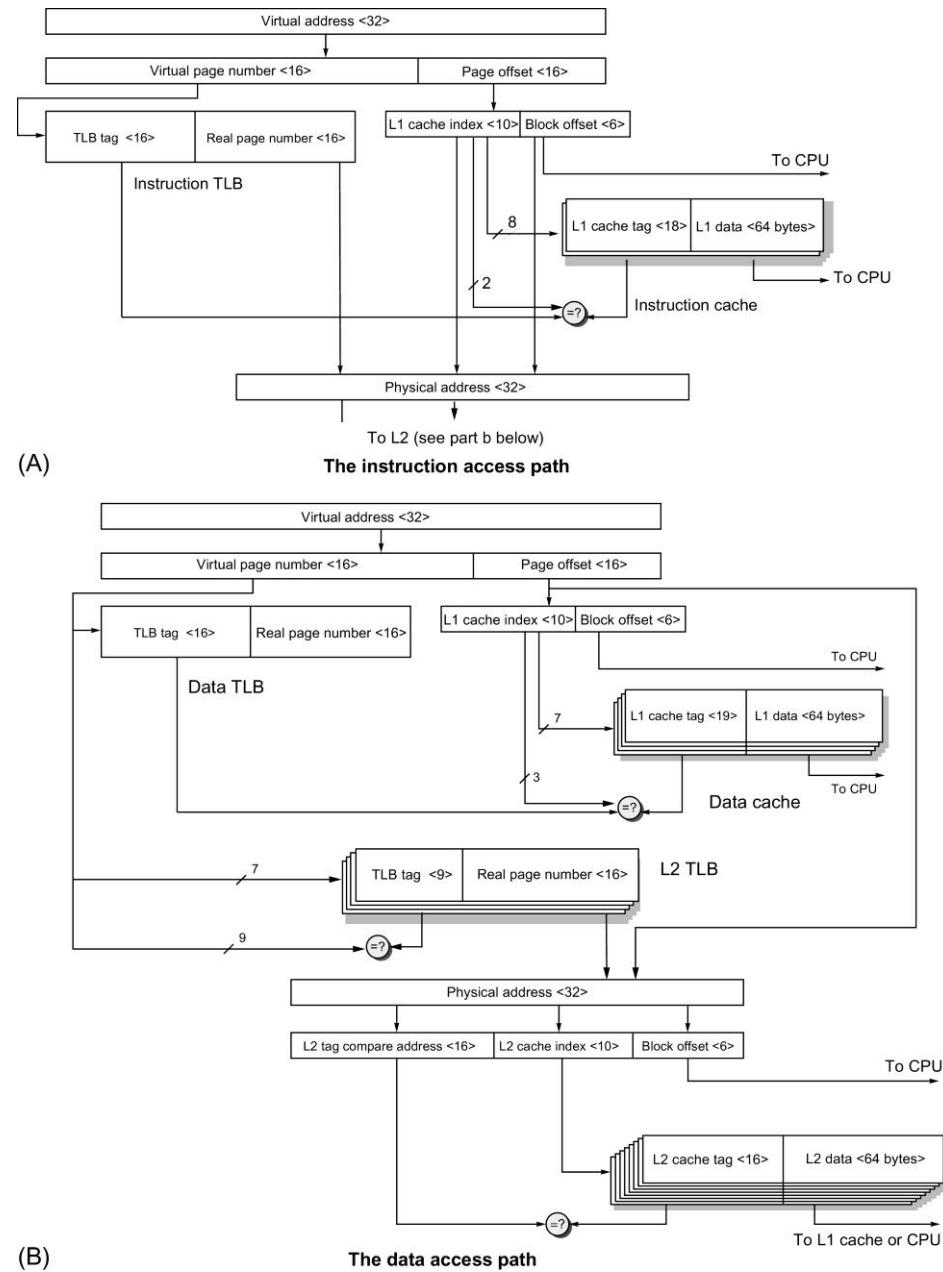
Figure 2.20 The virtual address, physical and data blocks for the ARM Cortex-A53 caches and TLBs, assuming 32-bit addresses. The top half (A) shows the instruction access; the bottom half (B) shows the data access, including L2.

The TLB (instruction or data) is fully associative each with 10 entries, using a 64 KiB page in this example.

The L1 I-cache is two-way set associative, with 64-byte blocks and 32 KiB capacity; the L1 D-cache is 32 KiB, four-way set associative, and 64-byte blocks.

The L2 TLB is 512 entries and four-way set associative.

The L2 cache is 16-way set associative with 64-byte blocks and 128 cKiB to 2 MiB capacity; a 1 MiB L2 is shown. This figure doesn't show the valid bits and protection bits for the caches and TLB.



Intel I7 TLB

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Entries	128	64	1536
Associativity	8-way	4-way	12-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	8 cycles
Miss	9 cycles	9 cycles	Hundreds of cycles to access page table

Figure 2.23 Characteristics of the i7's TLB structure, which has separate first-level instruction and data TLBs, both backed by a joint second-level TLB. The first-level TLBs support the standard 4 KiB page size, as well as having a limited number of entries of large 2–4 MiB pages; only 4 KiB pages are supported in the second-level TLB. The i7 has the ability to handle two L2 TLB misses in parallel. See Section L.3 of online Appendix L for more discussion of multilevel TLBs and support for multiple page sizes.

Intel i7 3-level cache hierarchy

Characteristic	L1	L2	L3
Size	32 KiB I/32 KiB D	256 KiB	2 MiB per core
Associativity	both 8-way	4-way	16-way
Access latency	4 cycles, pipelined	12 cycles	44 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

Figure 2.24 Characteristics of the three-level cache hierarchy in the i7. All three caches use write back and a block size of 64 bytes. The L1 and L2 caches are separate for each core, whereas the L3 cache is shared among the cores on a chip and is a total of 2 MiB per core. All three caches are nonblocking and allow multiple outstanding writes. A merging write buffer is used for the L1 cache, which holds data in the event that the line is not present in L1 when it is written. (That is, an L1 write miss does not cause the line to be allocated.) L3 is inclusive of L1 and L2; we explore this property in further detail when we explain multiprocessor caches. Replacement is by a variant on pseudo-LRU; in the case of L3, the block replaced is always the lowest numbered way whose access bit is off. This is not quite random but is easy to compute.

Intel i7 Memory Hierarchy

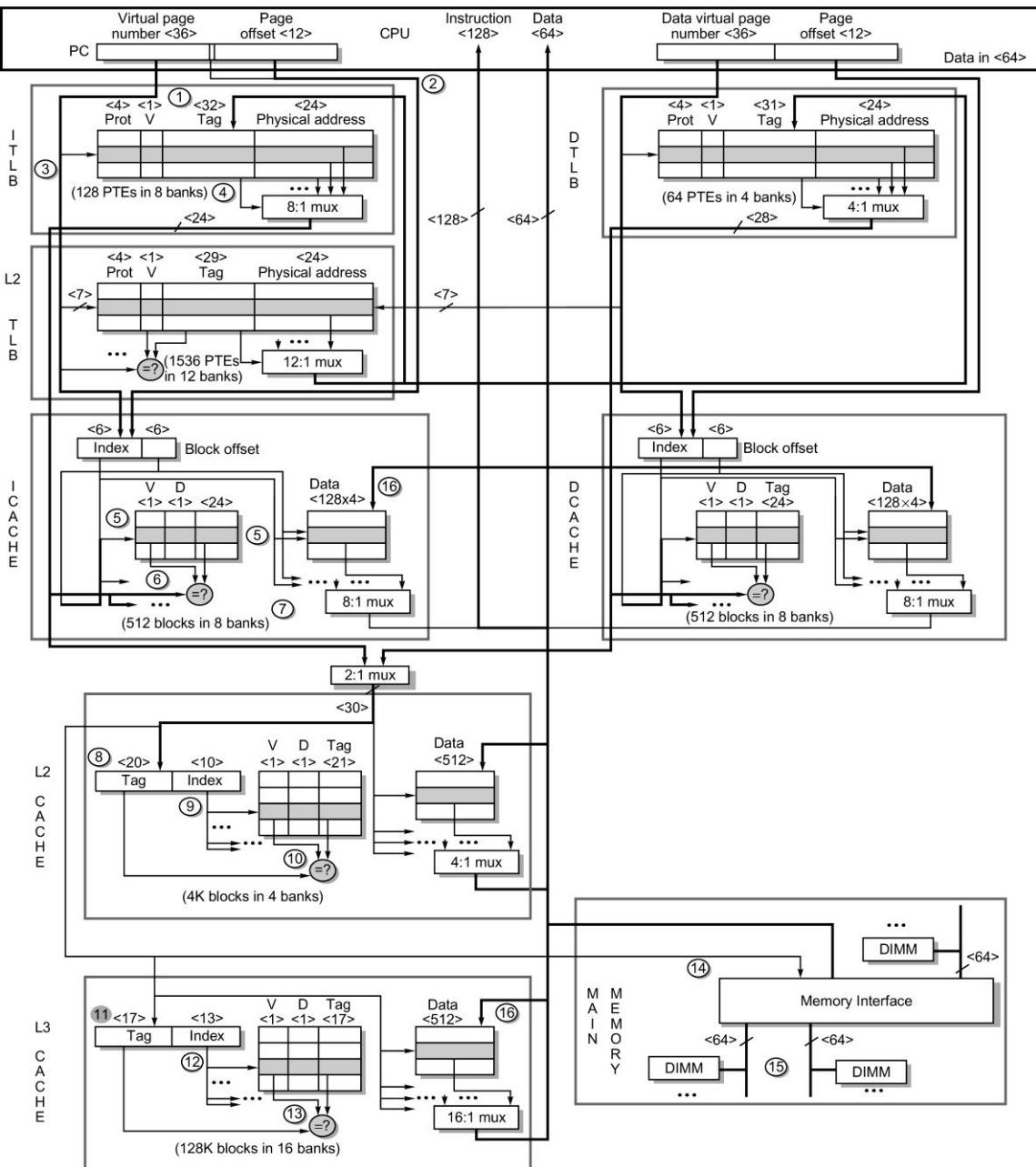


Figure 2.25 The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads for data. Writes are similar, in that they begin with a read (since caches are write back). Misses are handled by simply placing the data in a write buffer, since the L1 cache is not write allocated.

ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 15: Hardware Security: Physical Attacks

03/29/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Slides are adapted from Mark Tehranipoor, U. of Florida

Computer Security Landscape

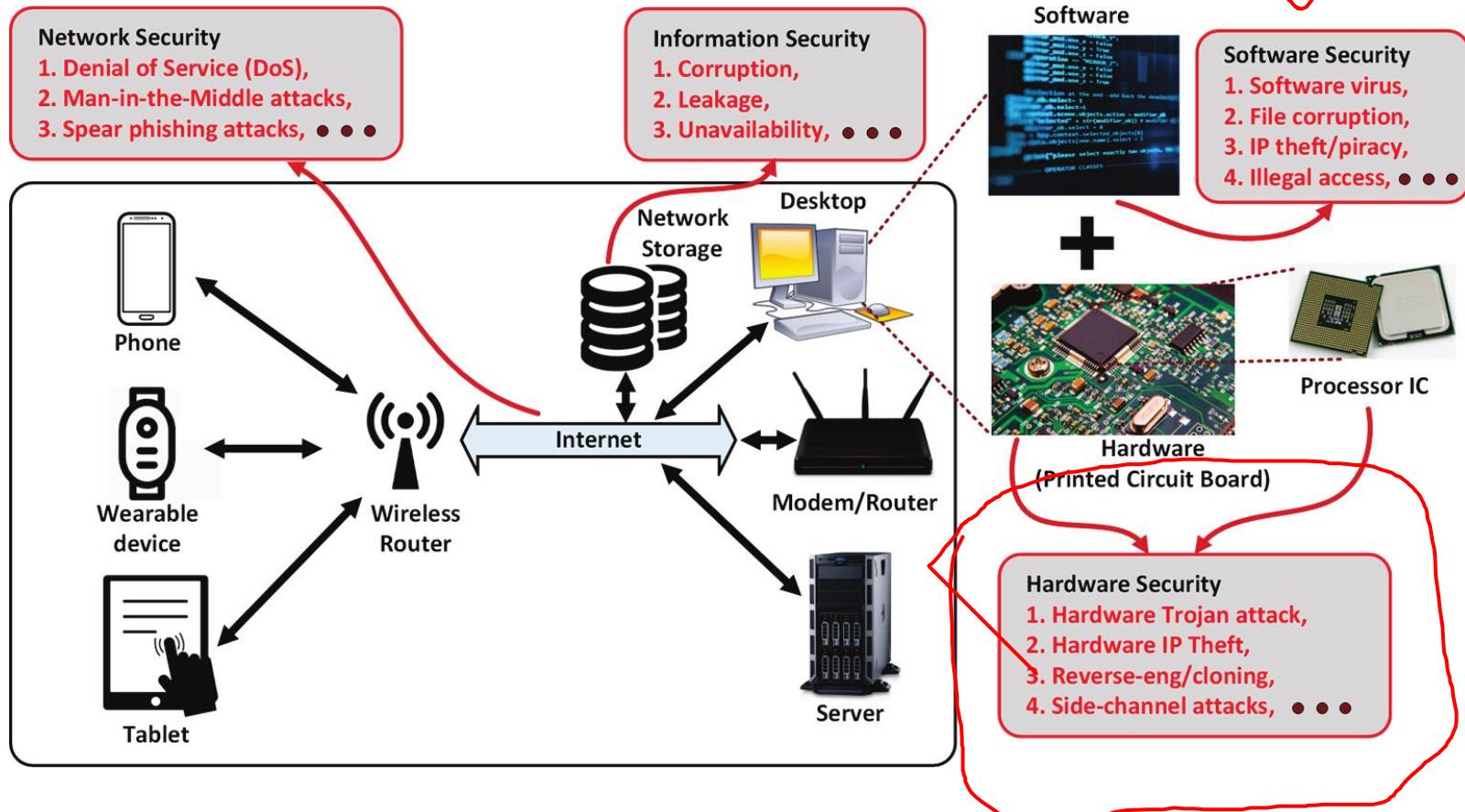


FIGURE 1.1 The landscape of security in modern computing systems.

Attack Impact and Difficulty

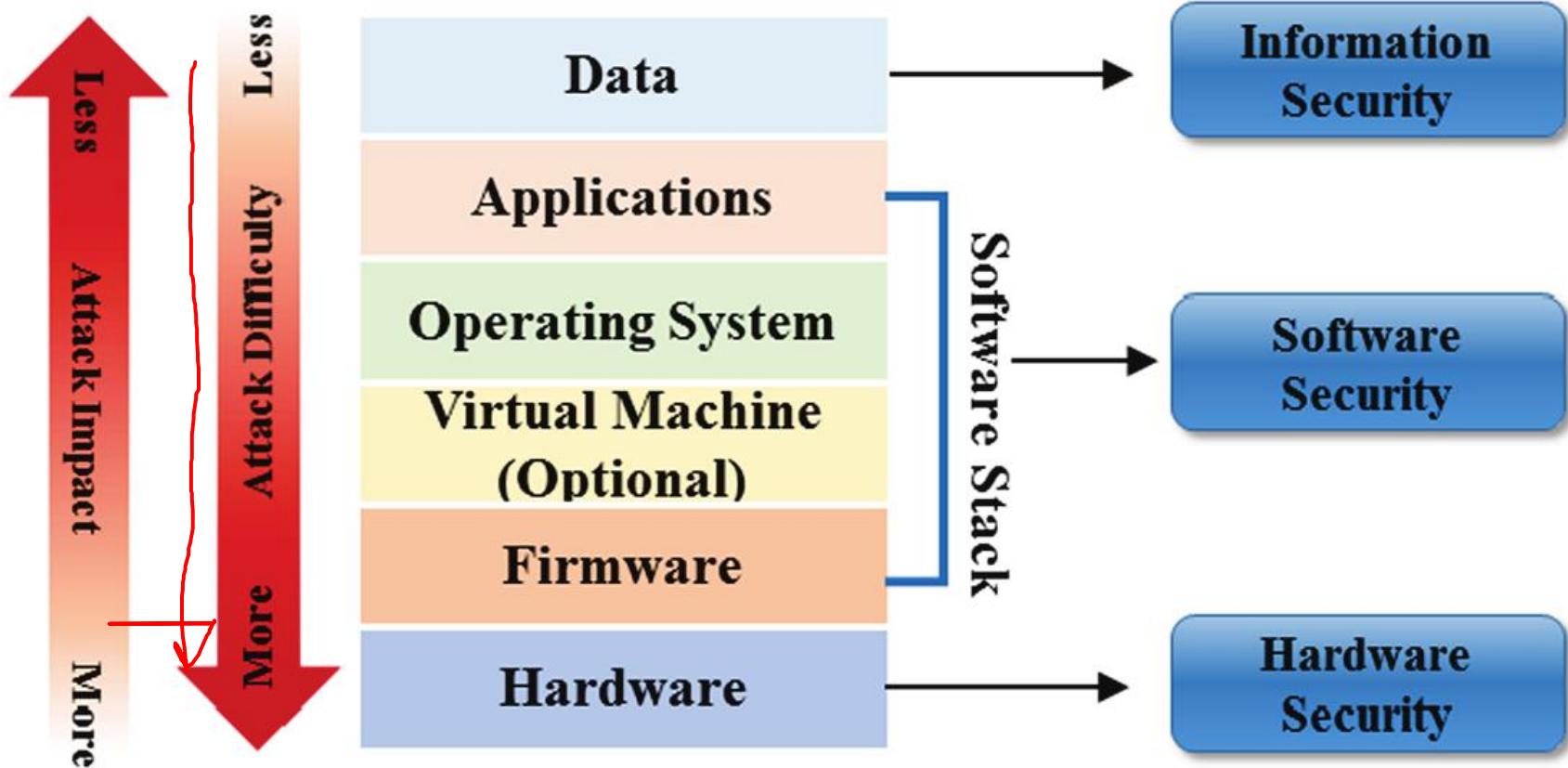


FIGURE 1.3 Attack impact and difficulty at different layers of a computing system.

Hardware Abstraction Layers

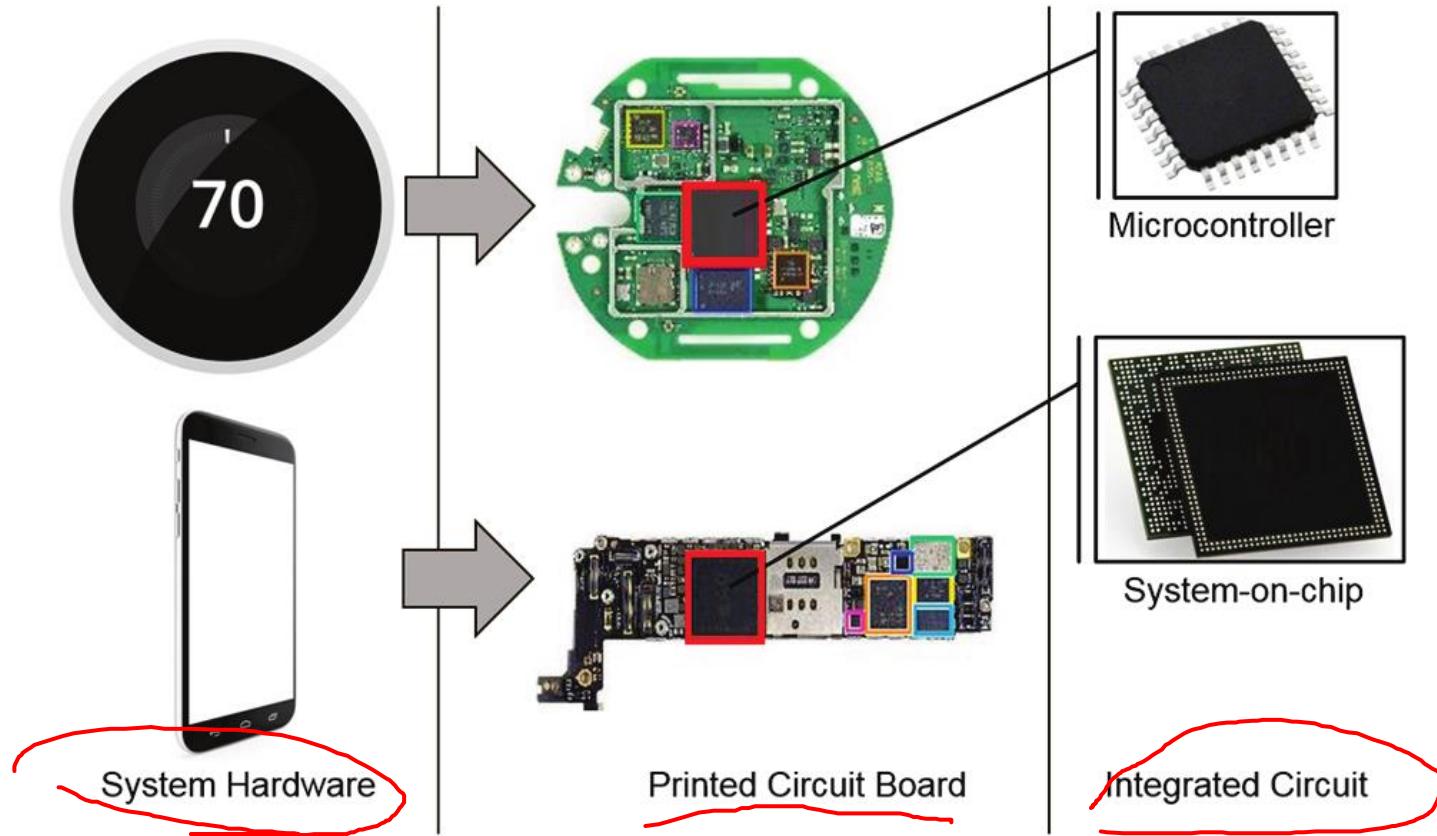


FIGURE 1.4 Three abstraction layers of modern electronic hardware (shown for two example devices).

Hardware Security and Trust

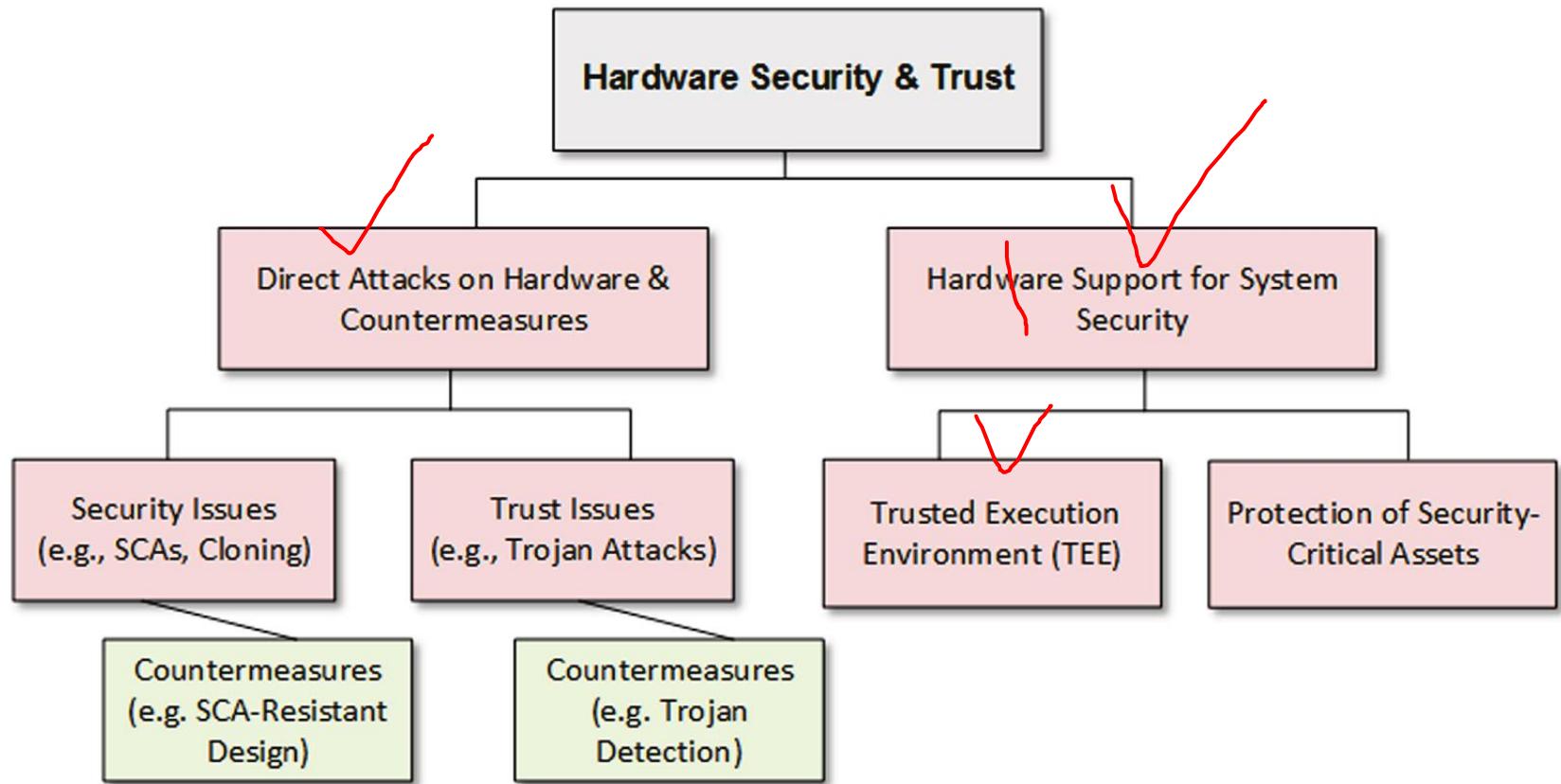


FIGURE 1.5 Scope of hardware security and trust.

Hardware Design and Test Flow

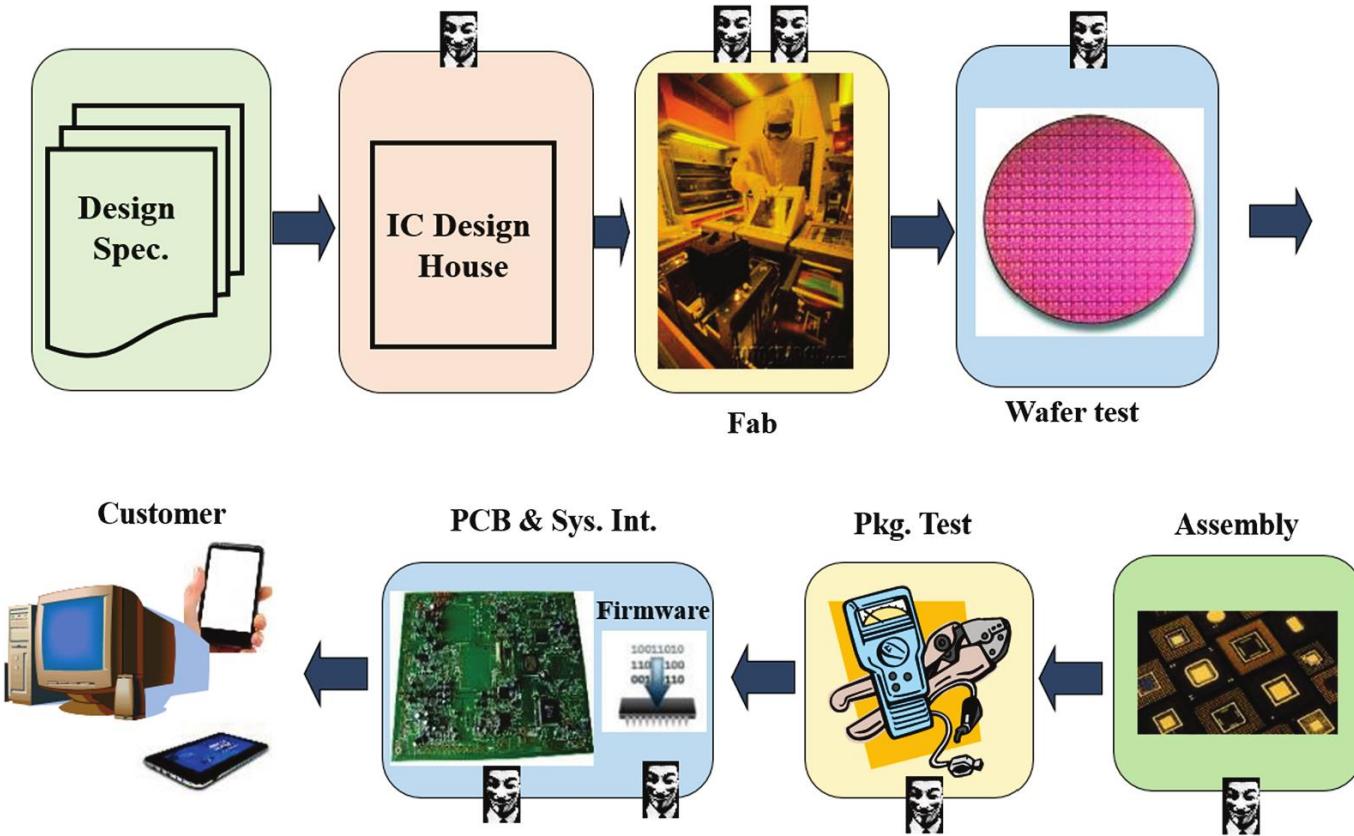


FIGURE 1.6 Major steps in the electronic hardware design and test flow.

Attack Vectors and Countermeasures

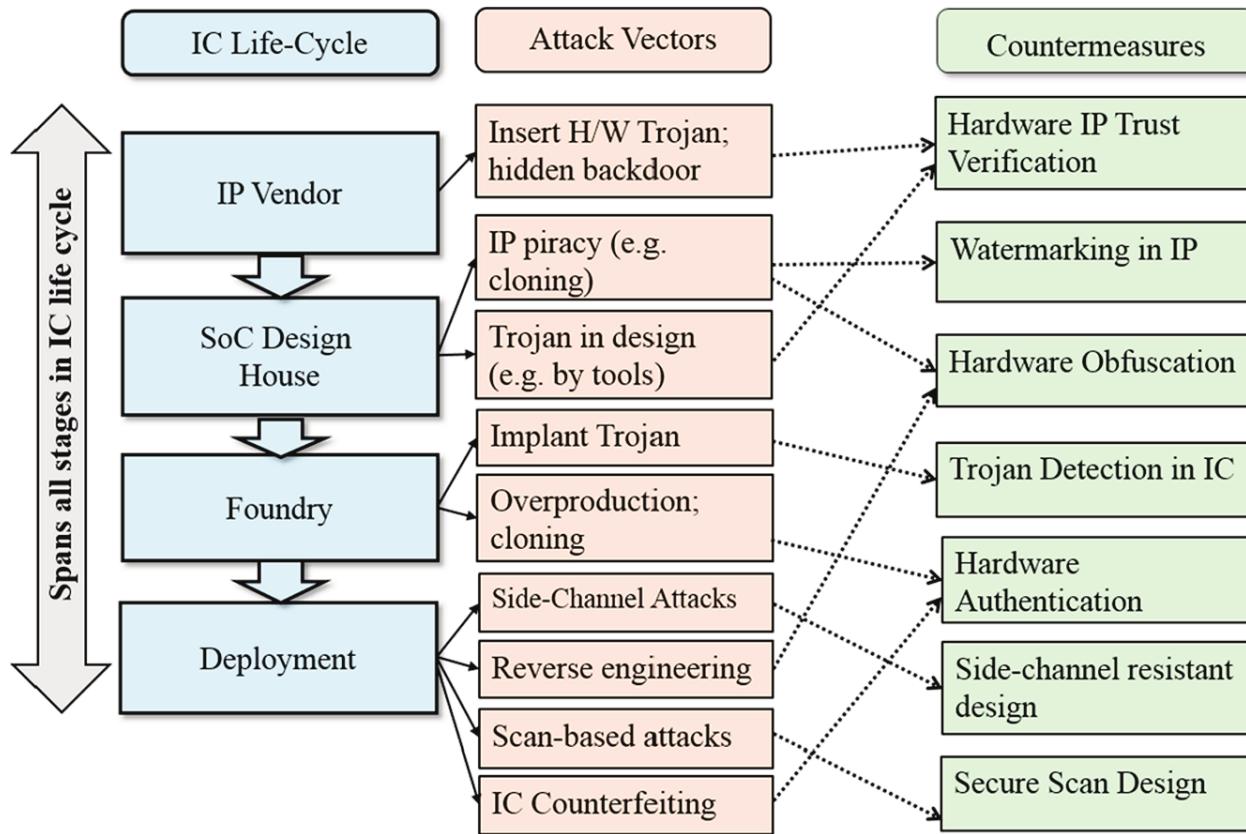


FIGURE 1.7 Attack vectors and countermeasures for each stage in an IC's life span.

Global Supply Chain Challenges

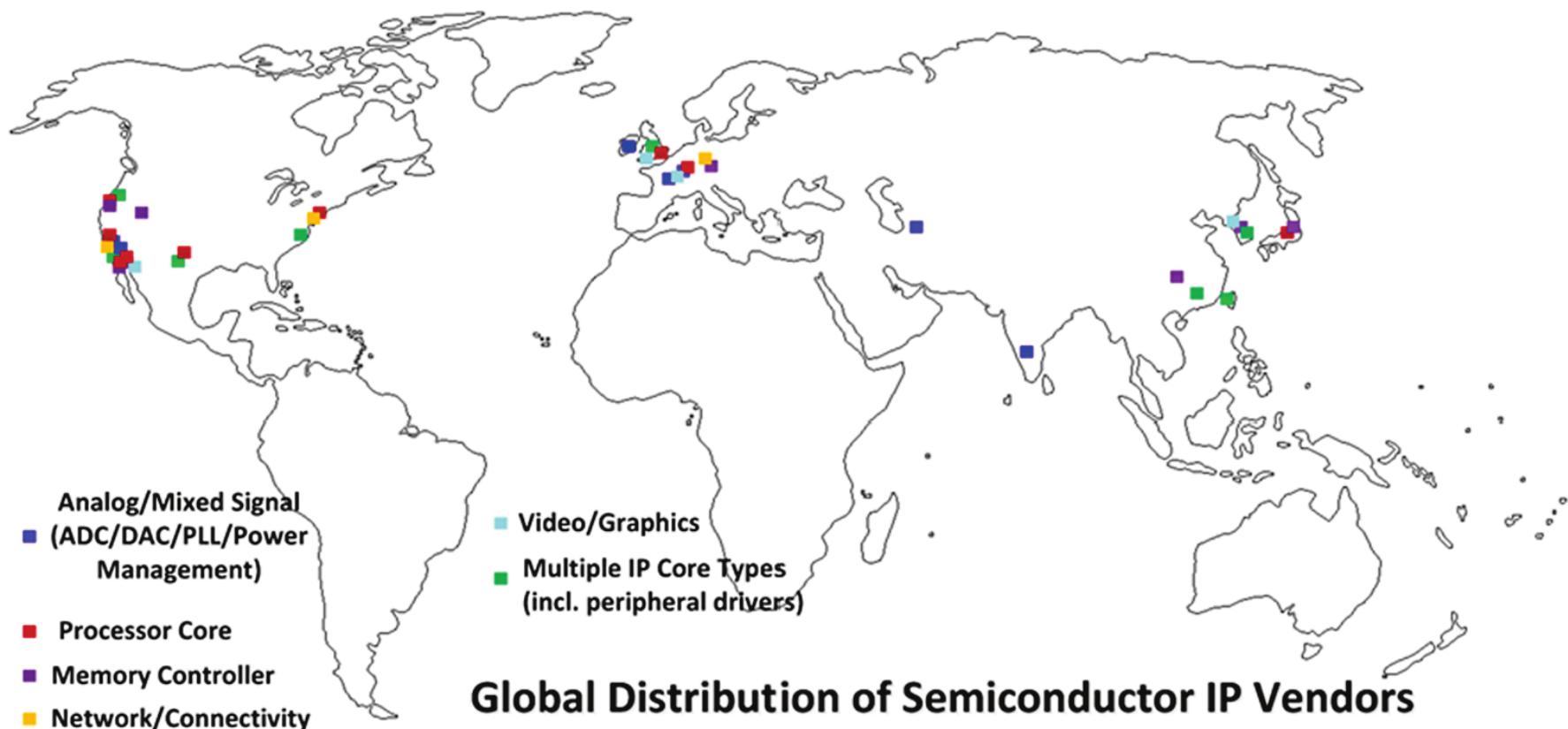


FIGURE 1.8 Long and globally distributed supply chain of hardware IPs makes SoC design increasingly vulnerable to diverse trust/integrity issues.

Attack Surfaces



FIGURE 1.9 Possible attack surfaces in a computing system.

Security Design and Validation

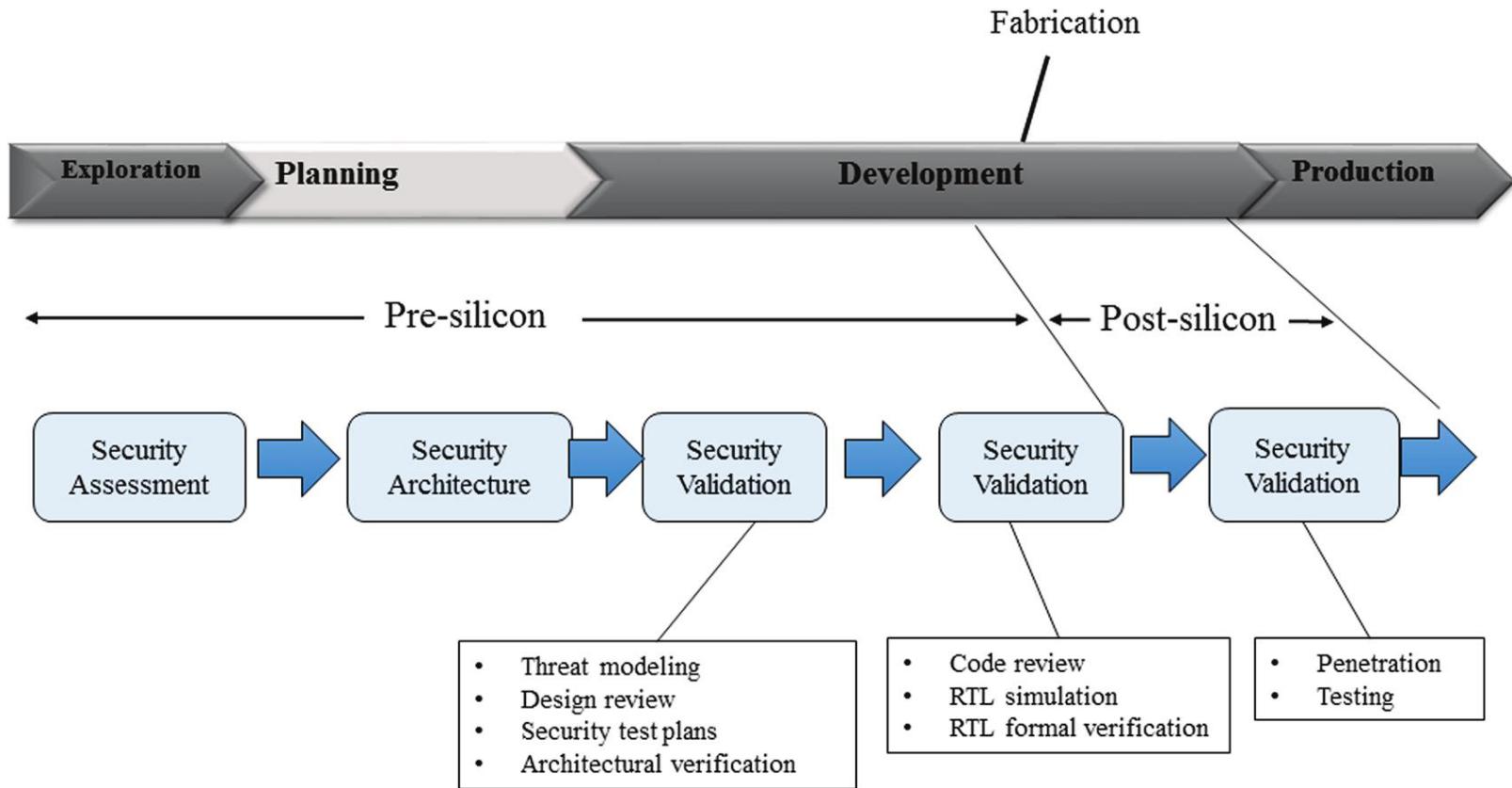


FIGURE 1.10 State of the practice in security design and validation along the life cycle of a system on chip.

Classification of Physical Attacks

Physical Attacks

Invasive Attacks

- Microprobing
- Reverse Engineering

Non-Invasive Attacks

- Side-channel Attacks
- Brute Force Attacks
- Fault Injection Attacks
- Data Remanence

Semi-Invasive Attacks

- UV Attacks
- Optical Fault Injection
- Advanced Imaging Techniques
- Optical Side-Channel Attacks

Non-Invasive Attacks

- Do not require *de-capsulation* or *de-layering* of the device, so it is non-destructive
 - Will not leave tamper evidence, so the user cannot be aware of the attack
- Do not require any initial preparation of the device under test
 - They can be done by tapping on a wire or plugging the device in the test chip.
- Easily reproducible, so they are not expensive
- It can take a lot of time to find an attack on any particular device.

Non-invasive attacks

Passive

- Side-Channel Attacks
 - Power Analysis Attacks
 - Timing Attacks
 - Electromagnetic Emission Attacks

Active

- Brute Force Attacks
- Glitch Attacks
- Under-voltage and over-voltage attacks
- Current Analysis

Invasive attacks

- Expensive to perform
 - require expensive equipment, knowledgeable attackers and sometime significant amount of time
 - almost unlimited capabilities to extract information from chips and understand their functionality
 - leave tamper evidence of the attack or even destroy the device
 - getting more demanding as the device complexity increases and the size shrinks (technology scales)
 - At the same time, the quality of the imaging devices is increasing

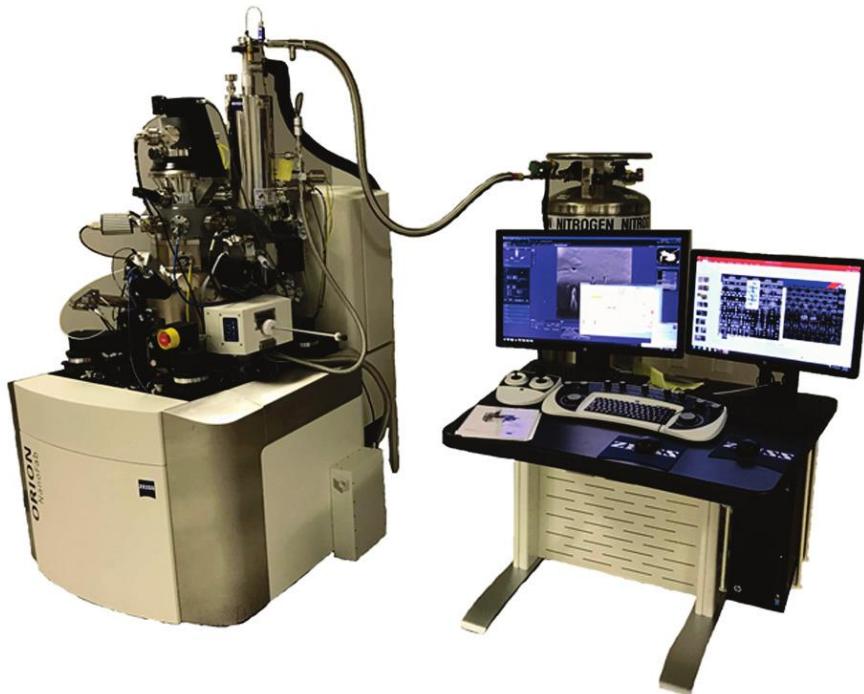
Invasive Attacks

- **Tools**

- IC soldering/desoldering station
- simple chemical lab and high-resolution optical microscope
- wire bonding machine, laser cutting system, microprobing station
- oscilloscope, logic analyzer, signal generator
- scanning electron microscope and focused ion beam (FIB) workstation



(A)



(B)



(C)

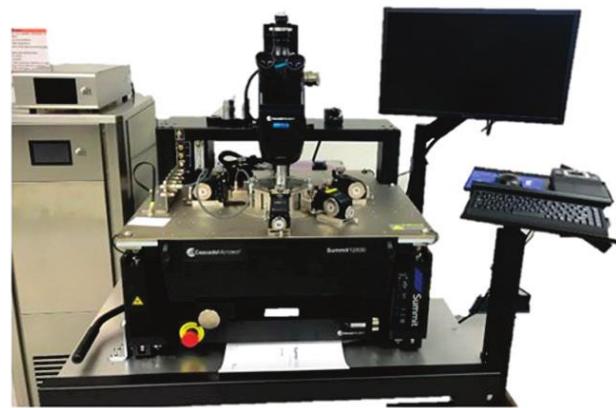
FIGURE 10.4 (A) Optical microscopy. (B) Scanning electron microscope (SEM). (C) Transmission electron microscope (TEM).



(A)



(B)



(C)

FIGURE 10.5 (A) Focused ion beam (FIB). (B) High-resolution x-ray microscopy. (C) Probe station.



(A)



(B)

FIGURE 10.6 (A) Logic analyzer. (B) Computer numerical control (CNC) [30xhtmGrizzly].

Semi-Invasive Attacks

- Relatively new type of attack, it fills the gap between *non-invasive* and *invasive* attacks
- Similar to the invasive attacks, they require de-packaging of the device
- The attacker do not need to have expensive tools such as FIB.
- Such attacks are not entirely new
 - E.g., UV light is used to disable security fuses in EPROM for many years

Semi-Invasive Attacks

UV Light Attacks

Used to disable security fuses in EPROM and one-time programmable (OTP) microcontrollers

Advanced Imaging Techniques

IR Light is used to observe the chip from rear side

Laser scanning techniques are used for hardware security analysis

Optical Fault Injection

It is used to induce transient fault in a transistor by illuminating it with laser

Optical Side-Channel Analysis

Observation of photon emission from the transistor

Invasive Attacks

Sample Preparation

Decapsulation

Deprocessing

Reverse Engineering

Optical imaging for layout reconstruction

Memory extraction

Microprobing

Laser cutter

FIB workstation

Chip Modification

FIB

Sample Preparation

- It starts with partial or full **decapsulation** of the chip to expose the chip die
- **Decapsulation** is the process of the removal of the chip package
 - It can be done easily by anyone who has low level chemistry knowledge
 - Only need to do some practice on a dozen chips

Manual Decapsulation

Milling a hole on the Chip Package

- In this way the acid will affect only desired area on the chip surface

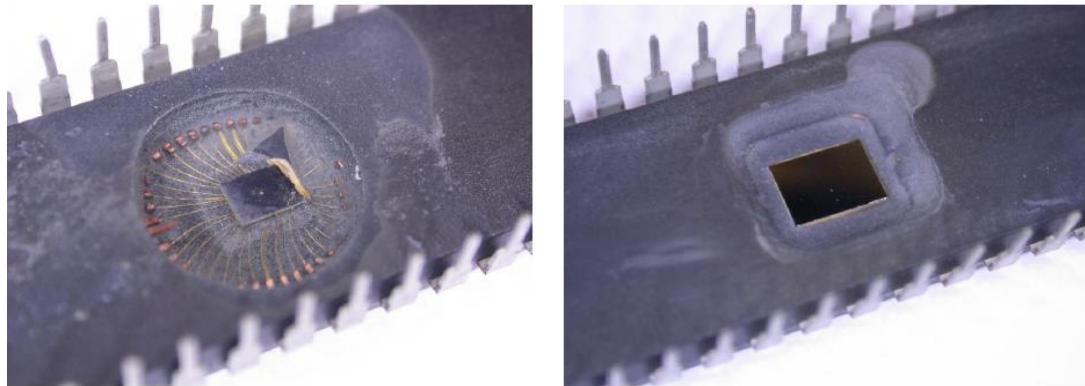
Exposing the chip package to acid

- Fuming Nitric Acid or mixture of Fuming Nitric Acid and concentrated Sulphuric Acid can be used
- The acid is applied with a pipette to the hole in the chip, it should be preheated to 50-70 °C

Cleaning the chip from the reaction products

- After 10-30 second, the chip is sprayed with dry acetone several times
- Also, ultrasonic bath can be used to clean the chip die surface

Manual Decapsulation



- **Decapsulation can be done from the rear side of the chip**
 - Access to the chip die can be established without using any chemical
 - It requires to mill down to the copper plate which can be then removed mechanically

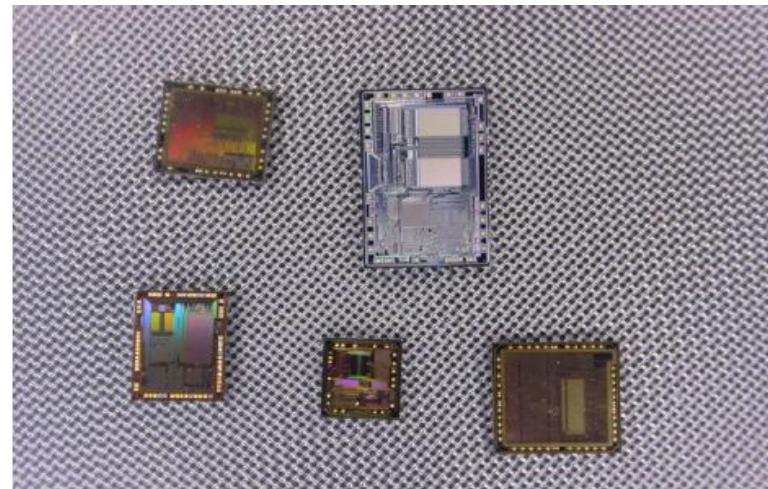
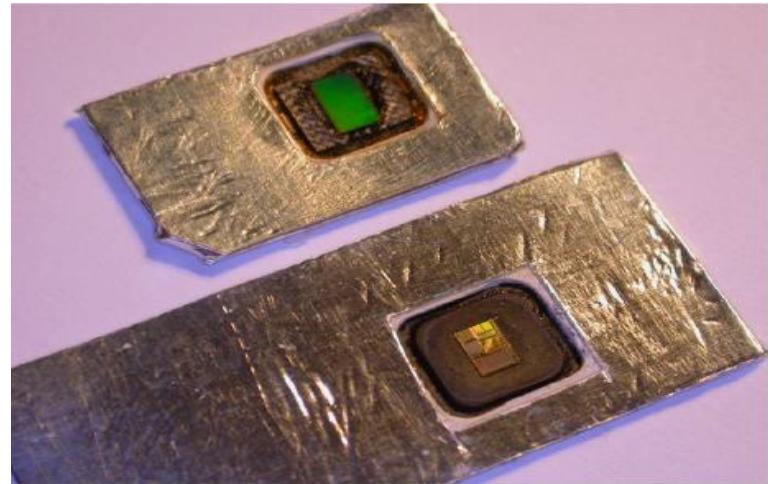
Automated Decapsulation

- For large quantities, automated decapsulation systems can be used.
 - Very little skill and experience is required to operate it
 - Cost around \$15,000
 - Also, they consume ten times more acid than the manual decapsulation, so the disposal of the waste should be done in proper way



Example Decapsulation

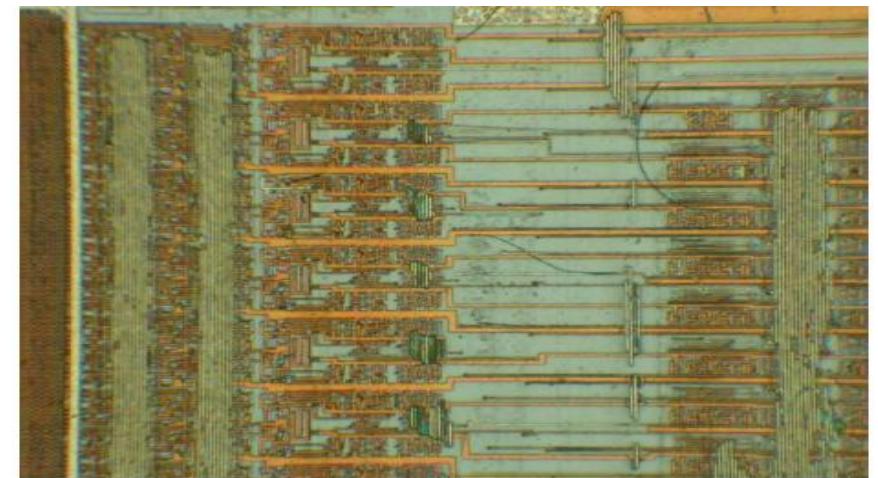
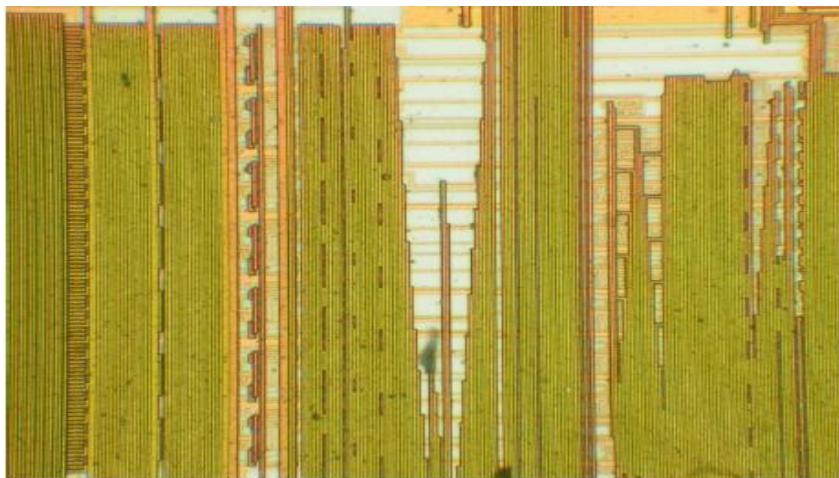
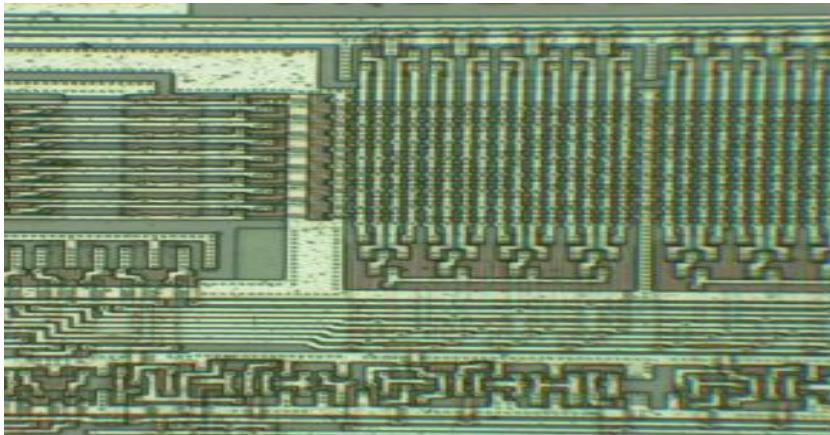
- The same partial decapsulation can be applied to smart card
- Not all of them may maintain their electrical integrity
- Generally, smart cards are decapsulated completely



Deprocessing

- **Deprocessing** is the opposite process of the chip fabrication
- It has two main applications:
 - Removing passivation layer to expose metal layers for microprobing attack
 - Gaining access to the deep layers to observe internal structure of the chip
- Three basic deprocessing methods are used:
 - Wet chemical etching
 - Plasma etching, also known as dry etching
 - Mechanical polishing

Deprocessing



Top: Motorola MC68HC705C9A microcontroller. The metal layer is removed exposing the polysilicon and the doping layers.

Bottom: Microchip PIC16F76 microcontroller. The top metal layer is removed exposing the second metal layer.

Reverse Engineering

- RE is used for understanding the **structure** of the device and its **functioning**
- For ASIC, it means locations of all the **transistors** and **interconnections**
- **All the layers** of the chip are removed one by one in reverse order and photographed to determine the internal structure of the chip
- Eventually, by processing obtained information, circuit netlist can be created and used to simulate the device

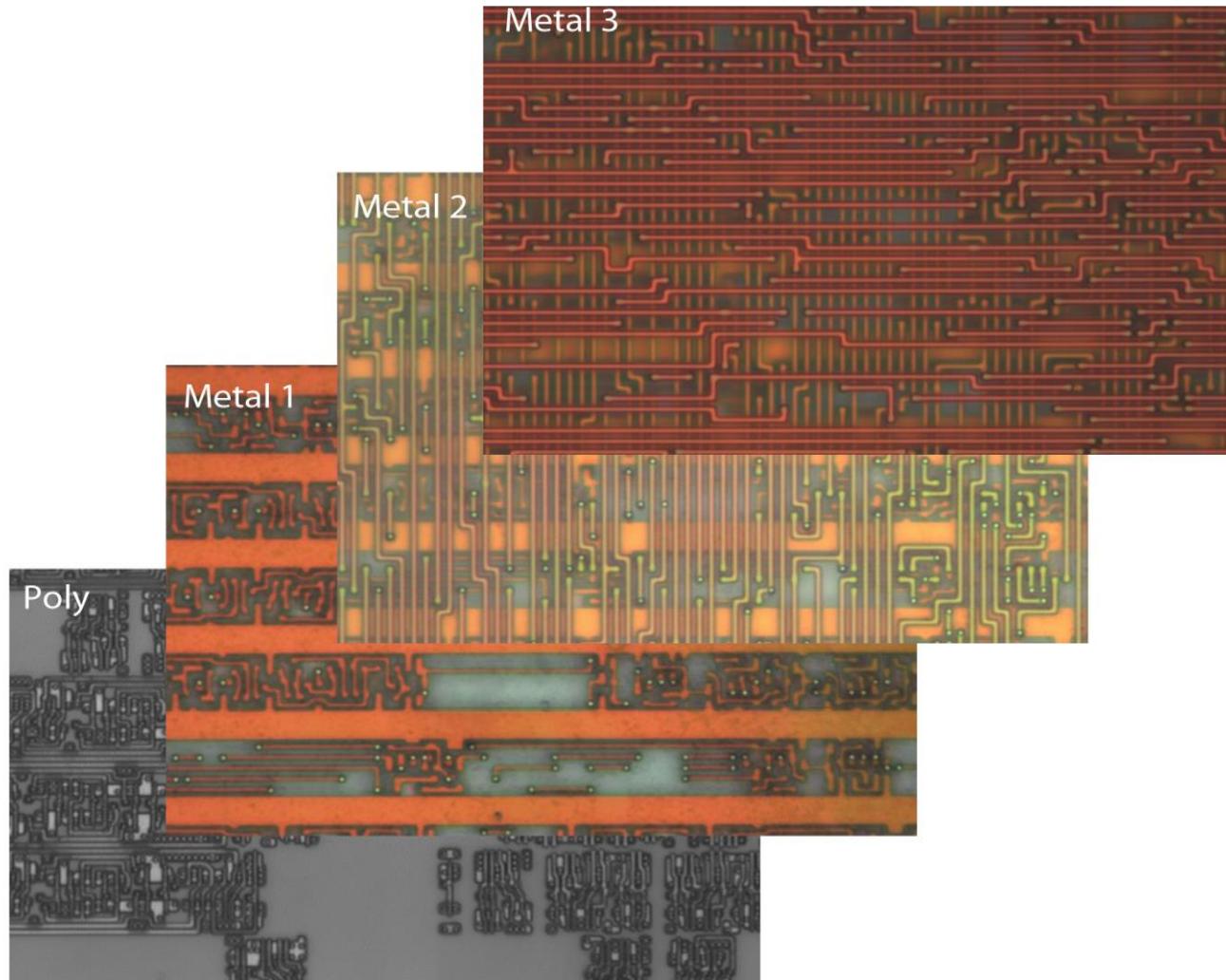
Reverse Engineering

- It is tedious and time-consuming process
- For the smartcards and microcontrollers, both **structural** and **program-code** reverse engineering is required.
 - First, security protection should be understood by **partial reverse engineering**
 - If memory bus encryption was used, the hardware responsible for this should be reverse engineered.
- For the CPLDs and FPGAs, even if the attacker obtained the configuration bitstream, he or she needs to spend a lot of time to simulate it

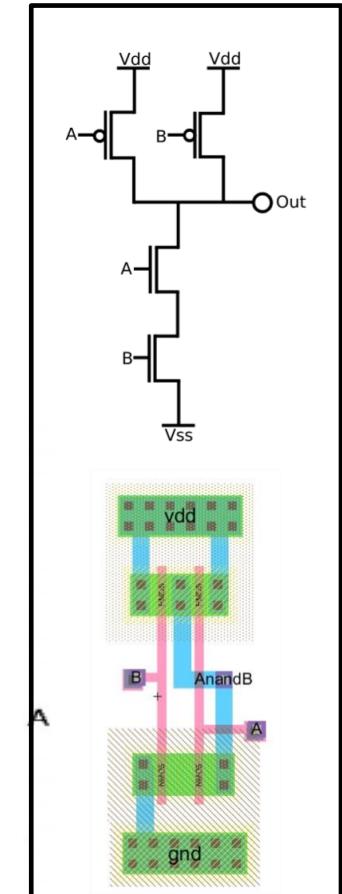
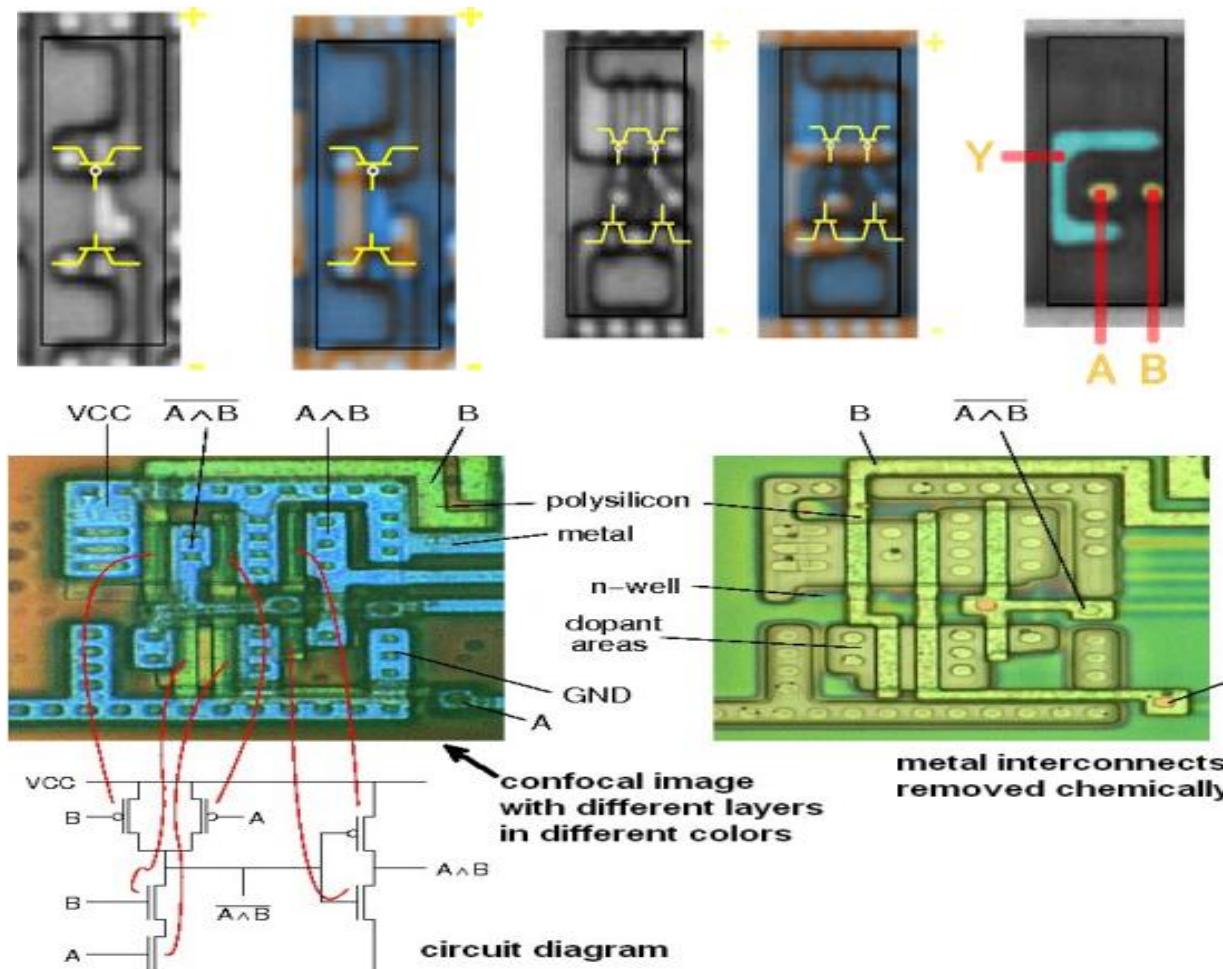
Reverse Engineering: Imaging

- **Optical Imaging:**
 - For reverse engineering the silicon chips down to $0.18 \mu\text{m}$ feature size, an optical microscope with a digital camera can be used
- **Scanning Electron Microscopy (SEM):**
 - For semiconductor chips fabricated with $0.13 \mu\text{m}$ or smaller technology, images are created using a SEM which has a resolution better than 10 nm .

Layer by Layer Imaging



Reverse Engineering



NAND Gate

Reverse Engineering: Memory Extraction

- **Memory Extraction from Mask ROMs**
 - Only possible for certain type of Mask ROM memory
 - NOR Mask ROM with active layer programming used in Motorola MC68HC705P6A Microcontroller can be read by removing the top metal layer
 - But, same Microcontroller with newer technology requires detailed deprocessing

Reverse Engineering: Memory Extraction

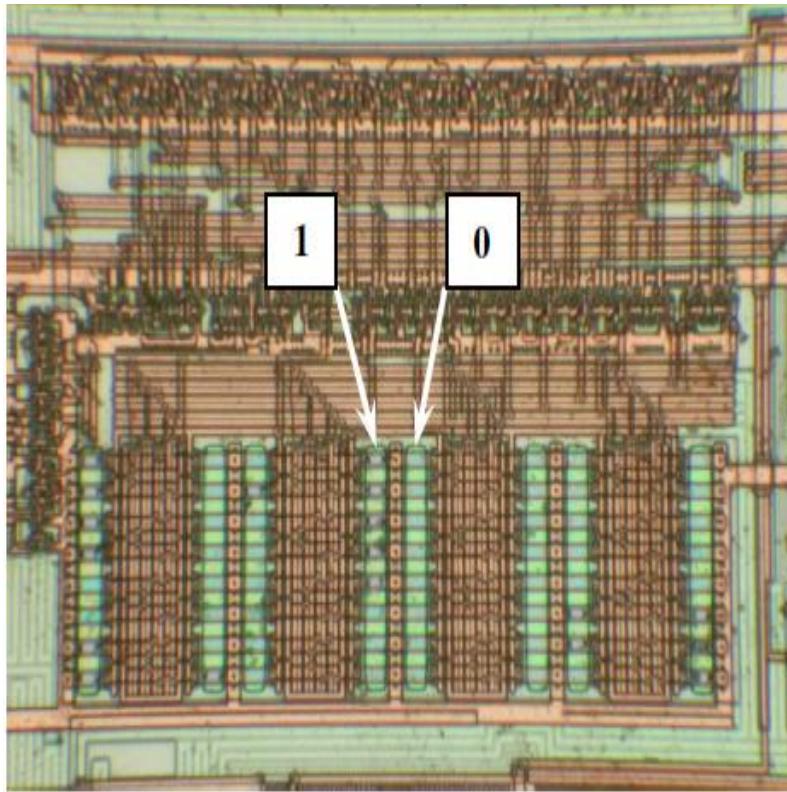


Figure 17. Laser ROM in Dallas DS1961S iButton chip [49]. Information can be read optically and altered with a laser cutter

The logic state is encoded by the absence or presence of the nor transistor OR the absence or presence of the a via plug from bit-line to the active area of a transistor.

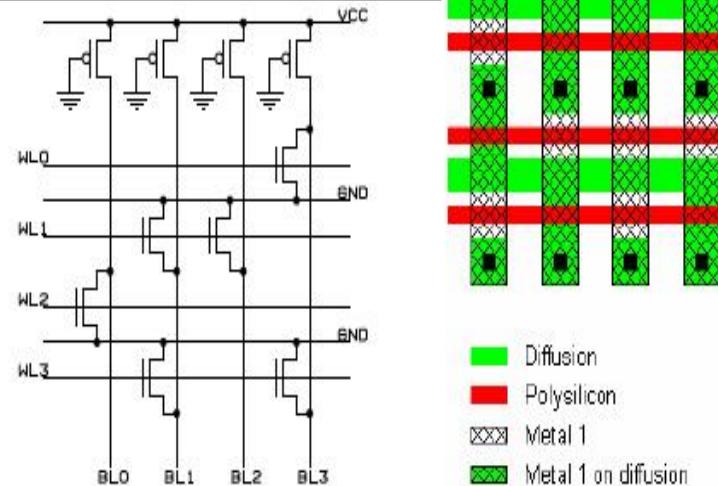


Figure 18. Configuration and layout of MOS NOR ROM with active layer programming. This type of memory can be read optically

Reverse Engineering

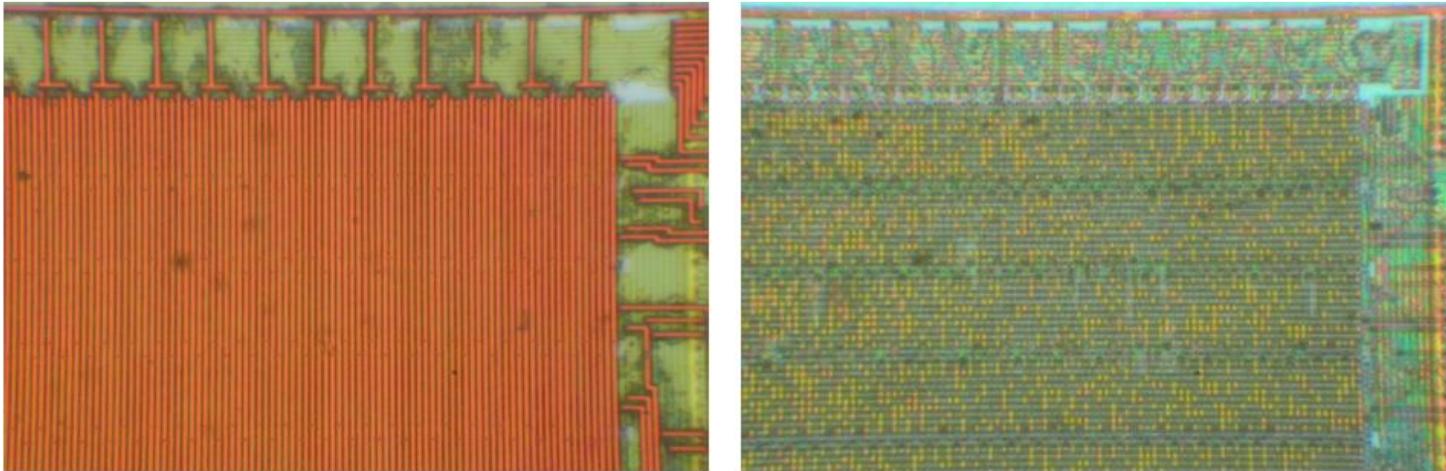


Figure 61. Optical image of the Mask ROM inside μ PD78F9116 microcontroller before and after wet chemical etching. 500x magnification

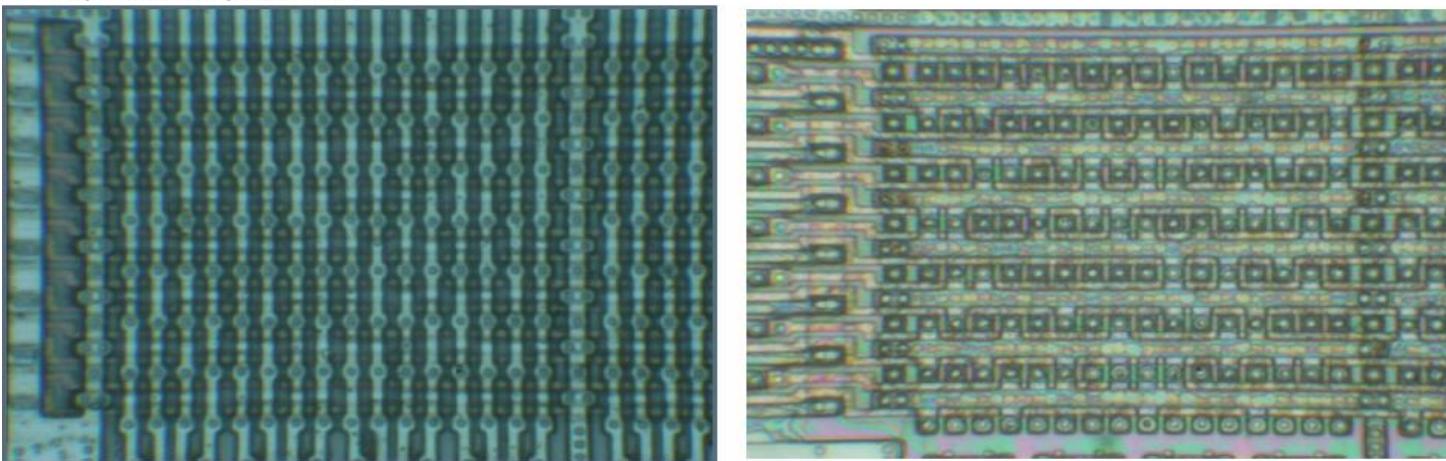


Figure 60. Optical image of the Mask ROM inside MC68HC705C9A microcontroller before and after wet chemical etching. 500x magnification

Reverse Engineering

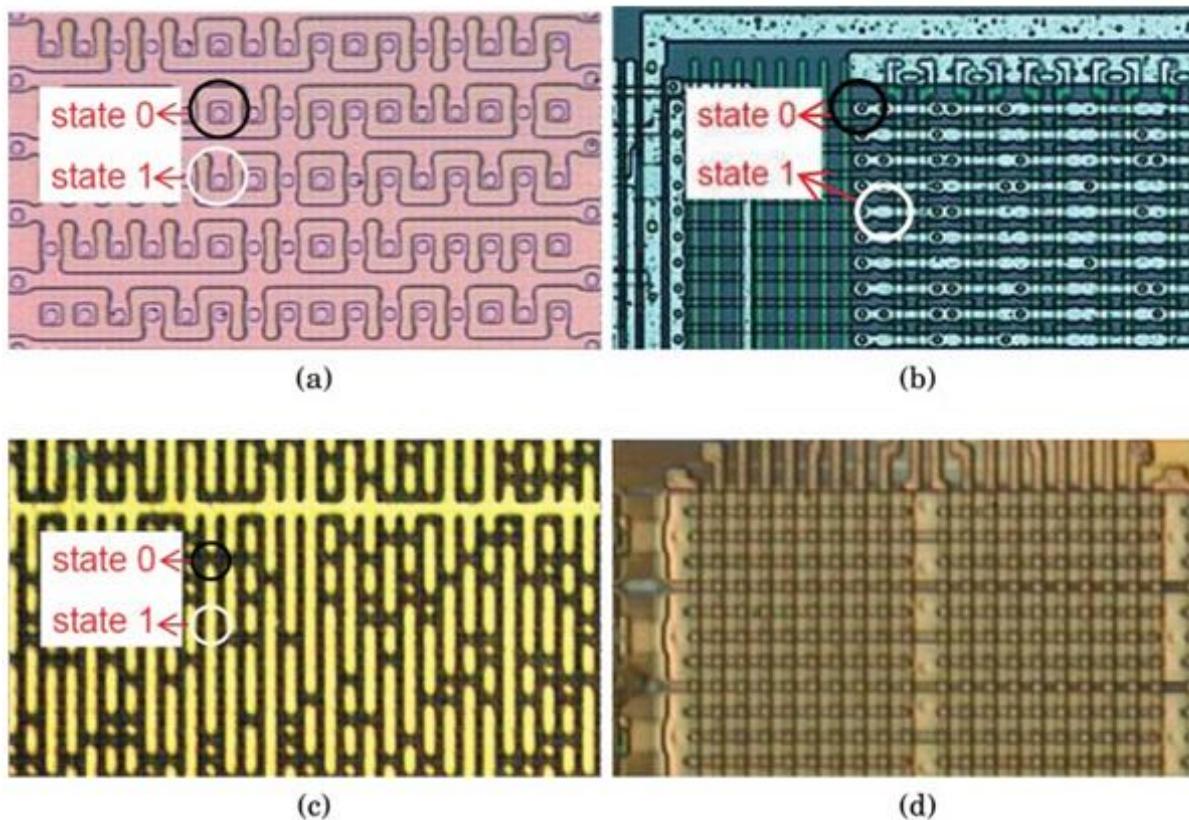


Figure 16: Optical inspection of active-layer programming ROM [63](a), contact- layer programming ROM [28] (b), metal-layer programming ROM [64](c), and implant pro- gramming ROM before selective etch [64](d).

Reverse Engineering

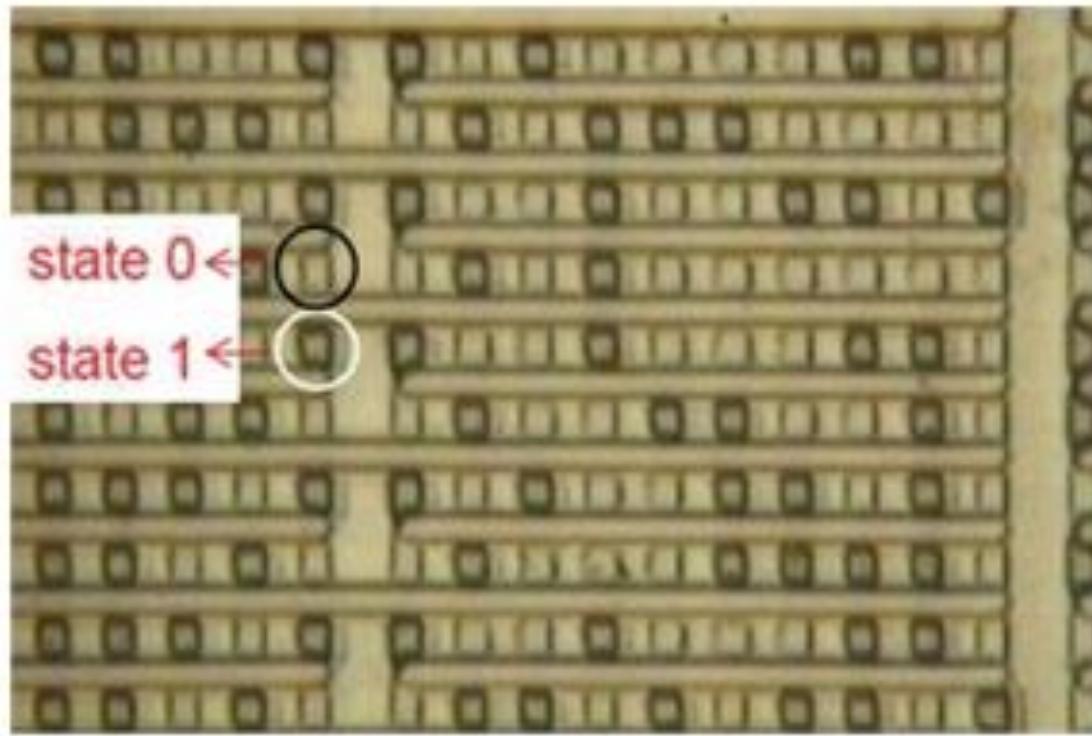


Figure 17: Optical inspection of implant programming ROM after selective etch [64].

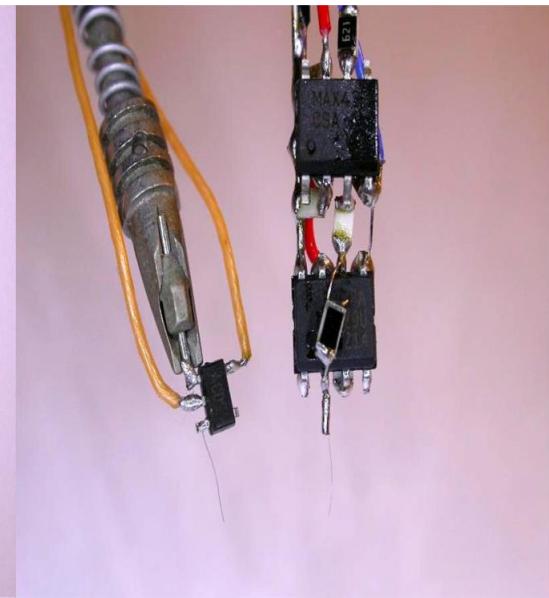
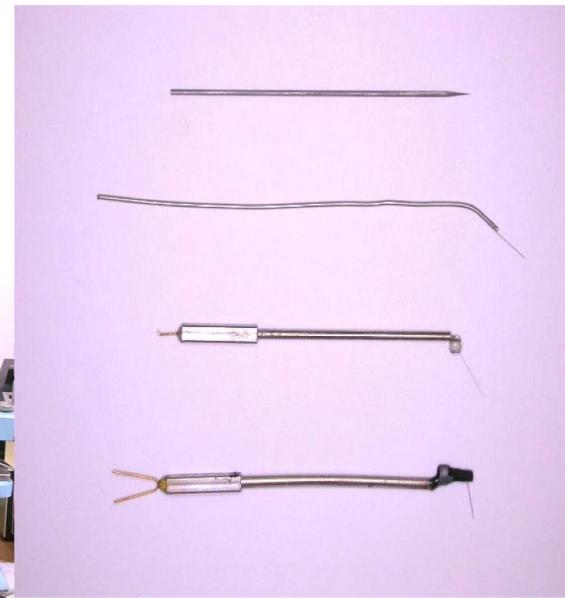
Invasive Attacks: Microprobing

- **Microprobing**
 - Could be used for both *Confidentiality* and *Integrity* violations
 - eavesdropping on signals inside a chip
(Confidentiality violation)
 - can be used for extraction of secret keys and memory contents
 - injection of test signals and observing the reaction
(Integrity violation)
 - laser cutter can be used to remove passivation and cut metal wires

Invasive Attacks: Microprobing

■ Tools

- The most important tool is microprobing station. It consists of five elements
 - a microscope, stage, device test socket, micromanipulators and probe tips.



Invasive Attacks: Microprobing

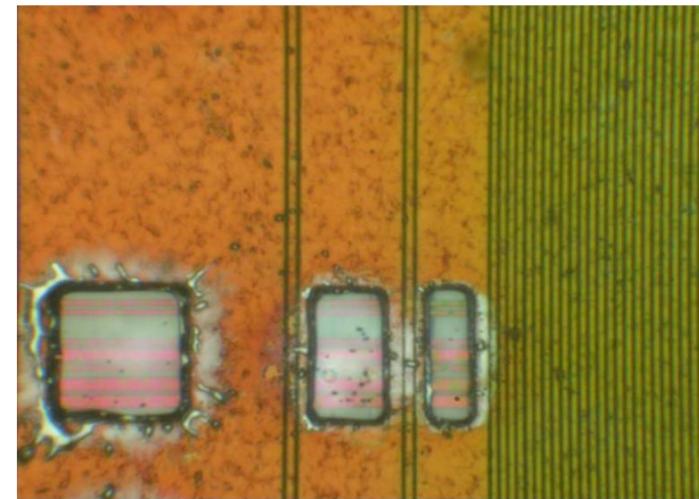
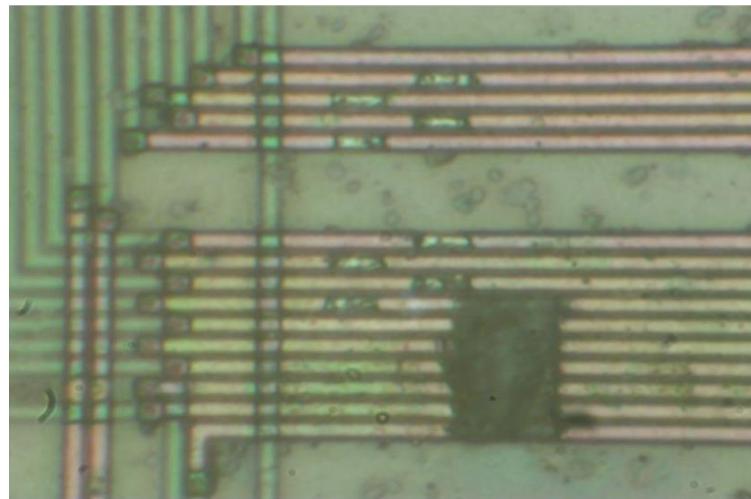
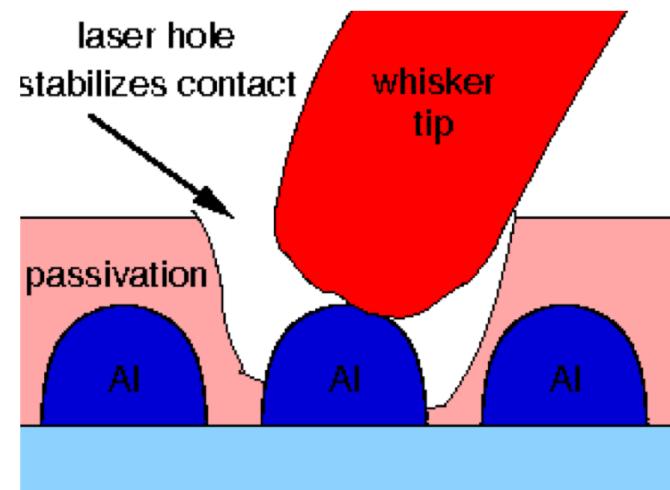
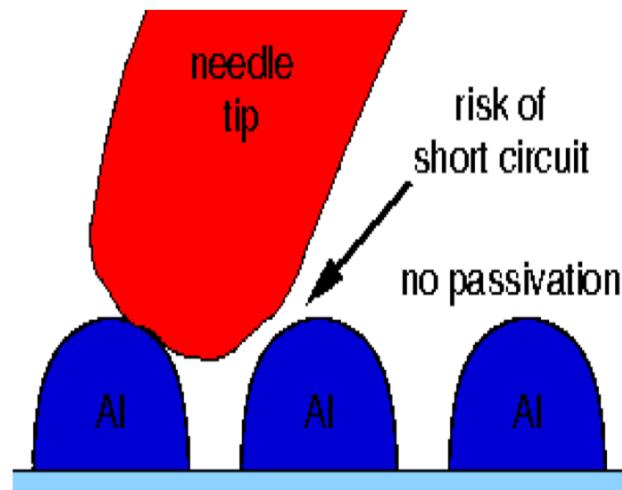
- **Microprobing is applied to the internal CPU data bus**
 - Difficult to observe whole data bus all at once
 - There are limited number of probes
 - Two to four probes are used to observe data signals which are combined as a whole data trace later.

Microprobing: Laser Cutting

- It is used to remove passivation layer to observe the metal layer
- Laser Cutting Systems consist of:
 - laser head mounted on camera port of a microscope
 - submicron-precision stage to move the sample
- Carefully dosed laser flashes remove patches of the passivation layer with micrometer precision



Microprobing: Laser Cutting



Microprobing: FIB Workstation

- The devices fabricated with lower technology node needs more sophisticated tools to establish contacts with the interconnect wires
- FIB stations can be used to create test point, imaging and repairing
- Also, FIB can mill holes and cut the wires



FIB Workstation

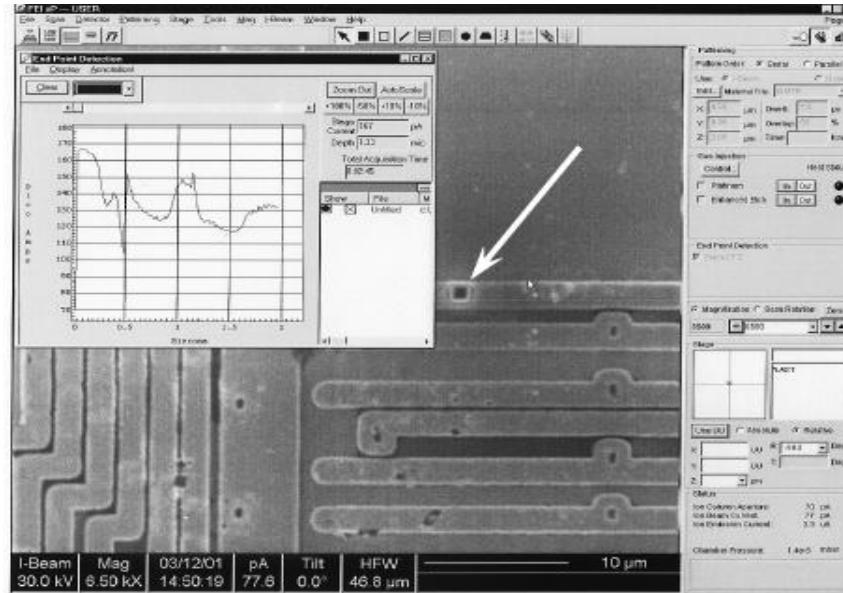


Figure 68. The process of milling the hole using FIB

A hole that is milled by FIB workstation. You can create really tiny holes on the chip die with FIB



Figure 69. Cutting the wires using FIB

wire cutting with FIB. It can be used for chip modification attacks to disable the security circuitry.

Invasive Attacks: Chip Modification

- It is used to disable security protection circuitry
 - By cutting one of the internal metal interconnection wires
 - By completely destroying the circuit associated with the security protection using a laser cutter
- For more sophisticated attacks FIB is used
 - Connecting the wire that transmits the security state to either the ground or the supply line.
- Chip modification always requires at least partial reverse engineering of the chip to find the point for possible attack.

Invasive Attacks: Chip Modification

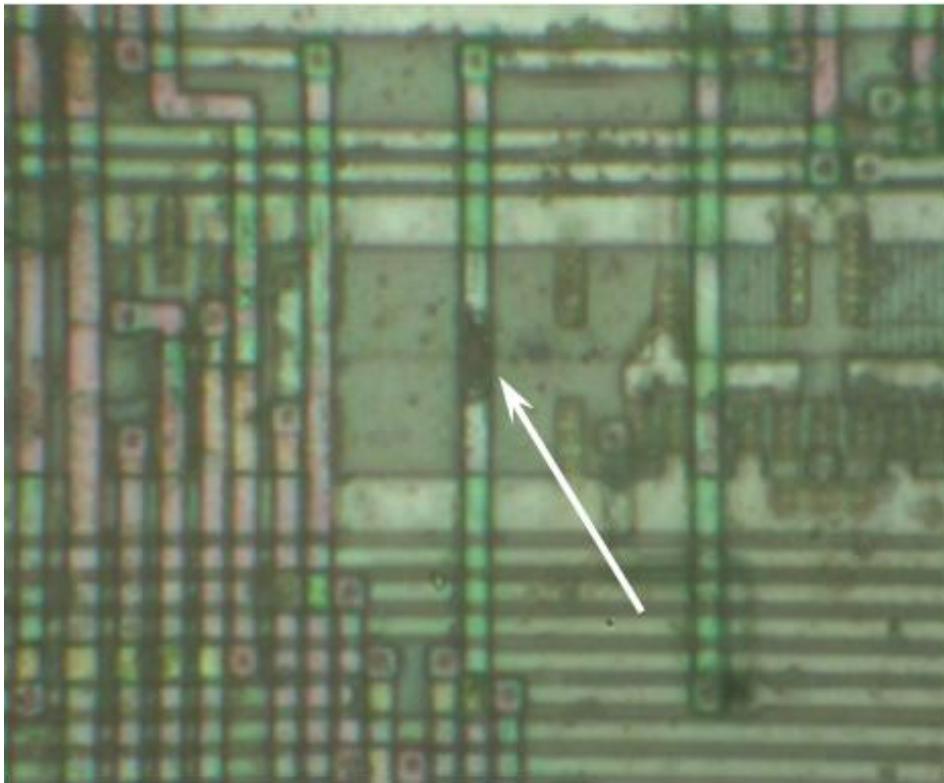


Figure 71. Cutting a single wire in the PIC12C508A microcontroller disables the security. 1000 \times magnification

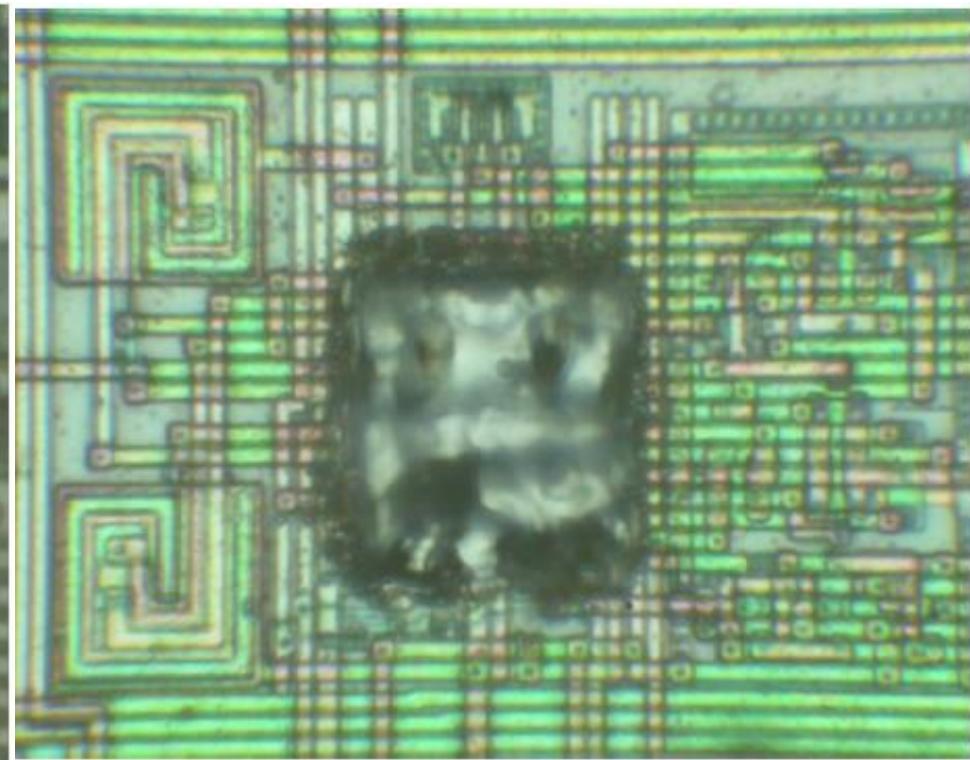


Figure 72. Disabling the security in the PIC16F628 microcontroller by destroying the fuse control circuit with a laser cutter. 500 \times magnification

Semi-Invasive Attacks

Sample Preparation

Decapsulation

Imaging

Backside imaging techniques

Perform the Attacks

UV light attacks

Active photon probing

Optical Fault injection attacks

Semi-Invasive Attacks: Imaging

- Down to 0.8 μm technology, it was possible to identify all the major elements of microcontrollers – ROM, EEPROM, SRAM, CPU
- Difficult to distinguish for newer technologies
- Can be observed with infrared light from rear side
- Backside imaging also is useful to extract the Mask ROM content

Semi Invasive Attacks: Imaging

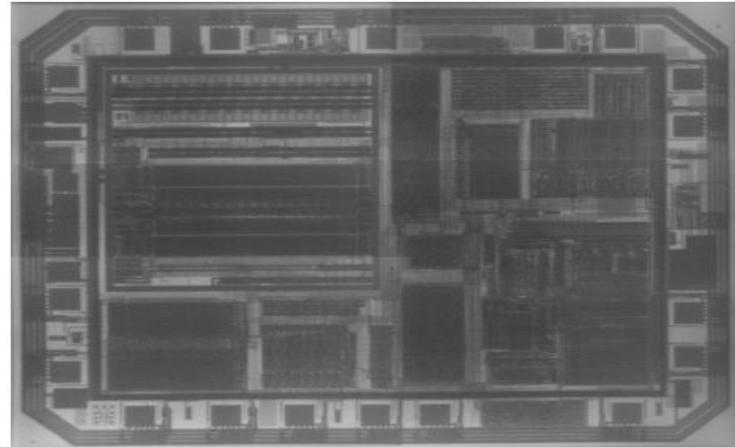
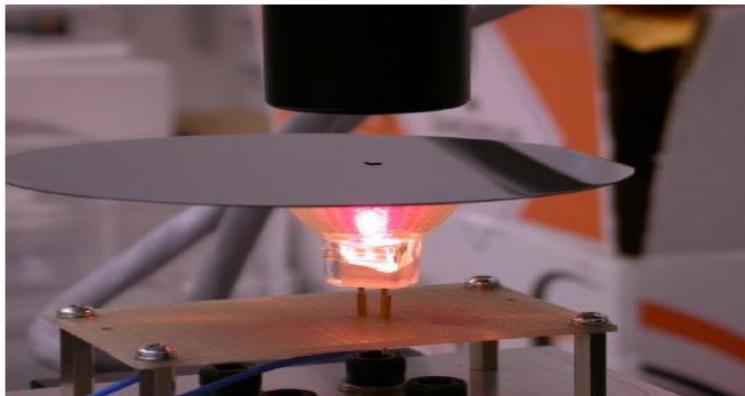


Figure 78. Transmitted light setup and image of the MSP430F112 microcontroller. 50x magnification

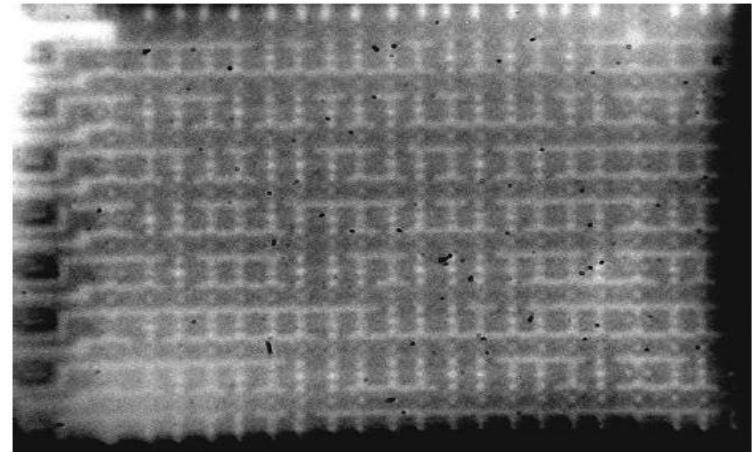
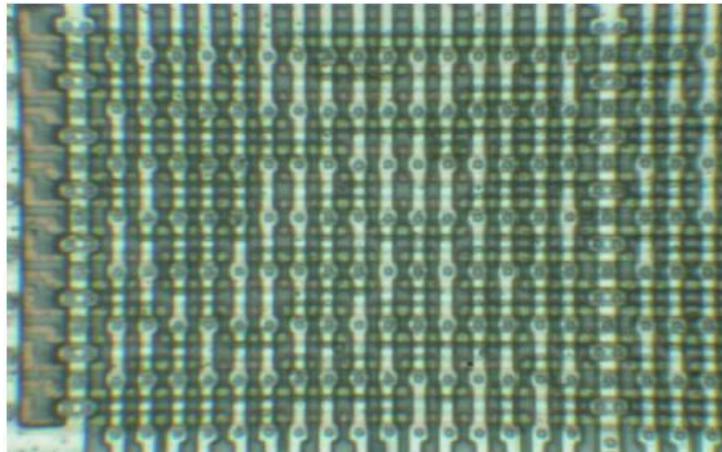


Fig 12 March 2018 Standard optical image and reflected light backside image of the Mask ROM inside MC68HC705D2A microcontroller built with 1.0 μm technology. 500x magnification

Semi-Invasive: Optical Fault Injection Attacks

- Illumination of a target transistor causes it to conduct, thereby inducing a transient fault
- Such attacks are:
 - Practical
 - Do not require expensive laser equipment
 - Any individual bit of SRAM in microcontroller can be set or reset

Fault injection attacks: Changing SRAM content

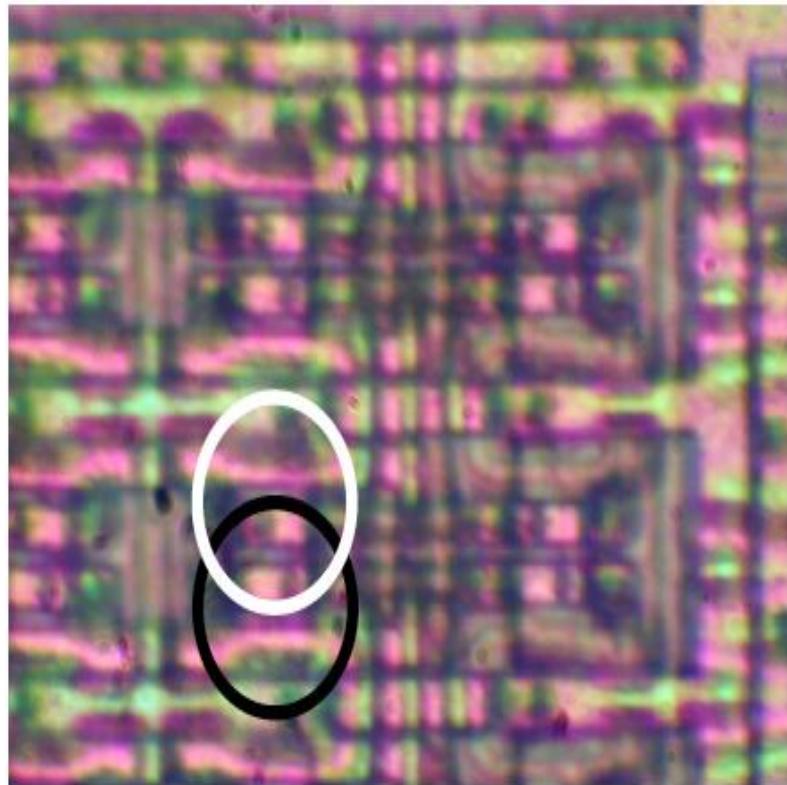


Figure 91. SRAM memory array with maximum magnification (1500x)

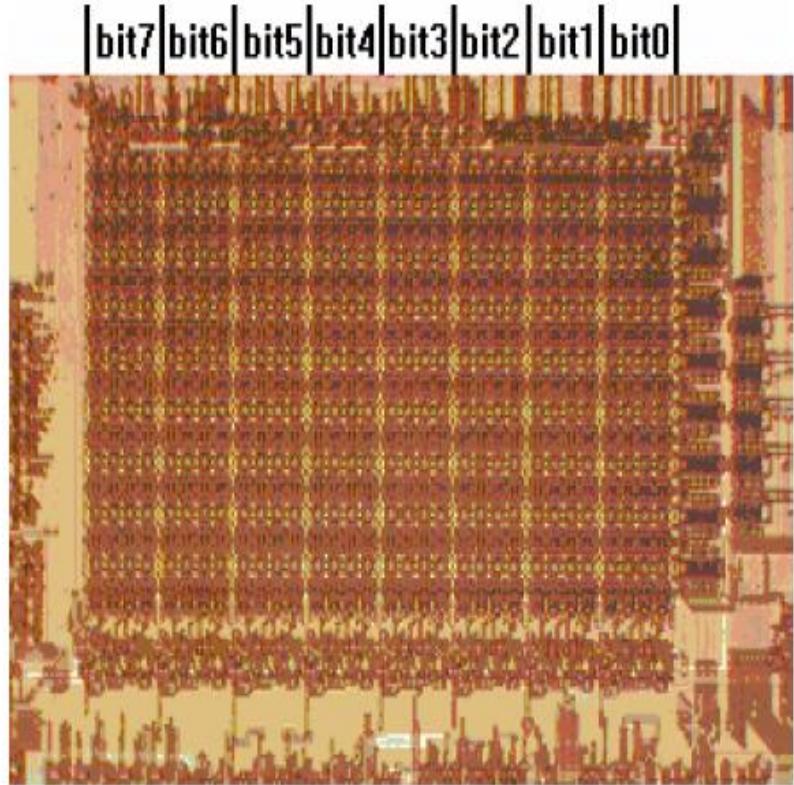


Figure 92. Allocation of data bits in SRAM memory array

Non-volatile memory contents modification

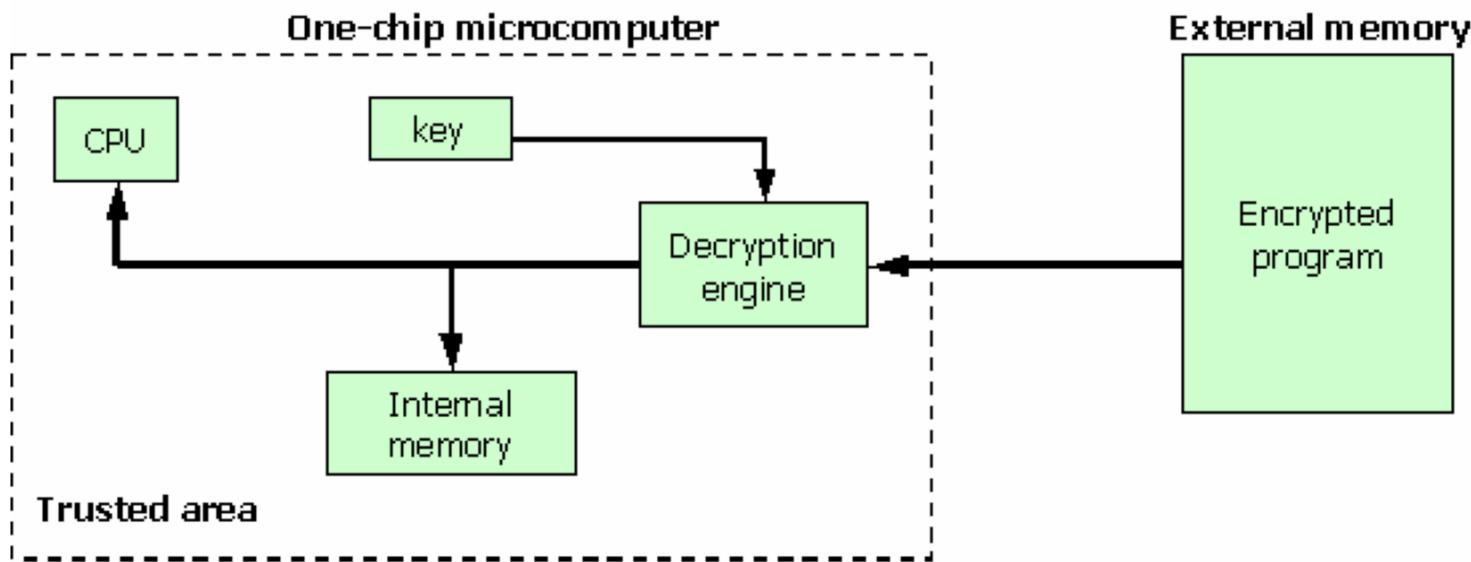
- EPROM, EEPROM and Flash memory cells are even more sensitive to fault injection attacks.
- They can be changed by light
- These attacks can be used to disable security fuses
 - The light should be focused down to the security fuse
- These attacks do not work on modern chips built in smaller sizes

Countermeasures

- Bus Encryption
- Top-layer Sensor Meshes
- ASICs and custom ICs
- Internal Voltage and Clock Frequency Sensors
- Light Sensor

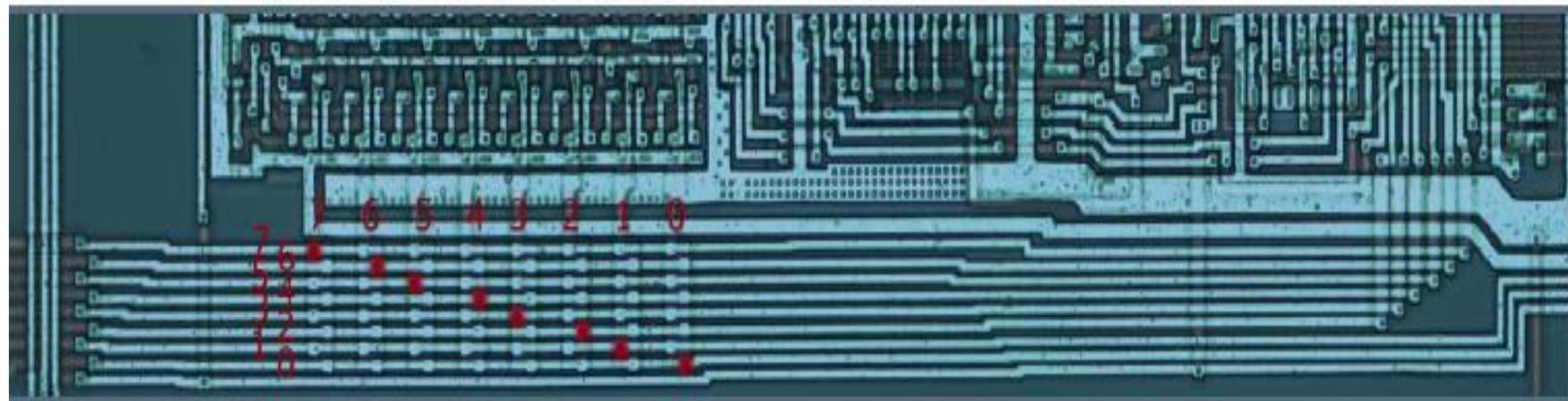
Countermeasures: Bus Encryption

- The **bus encryption** is used to protect the sensitive information from probing
 - Basically, the memory content is encrypted and then sent to the CPU by data bus
 - Before the data used in CPU, it is decrypted



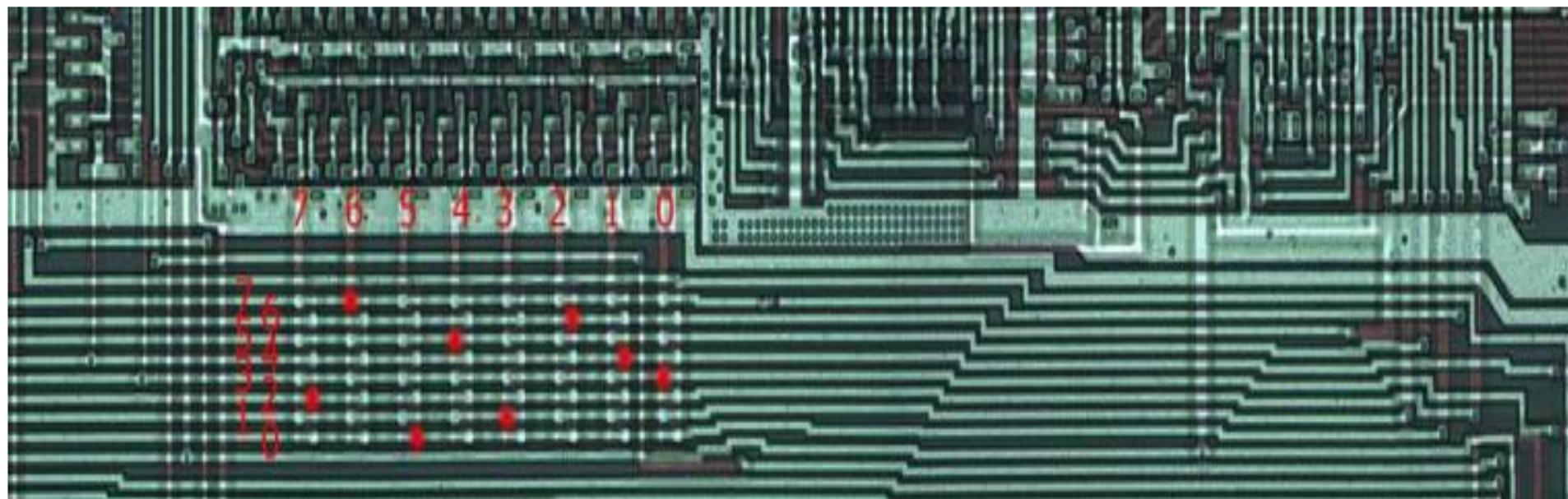
Countermeasure: Bus Scrambling

- **Typical probing areas**
 - Memory bus drivers
 - Data bus itself where lines are organized in proper CPU bus width
 - Bus order is always in order (0..7 or 7..0)



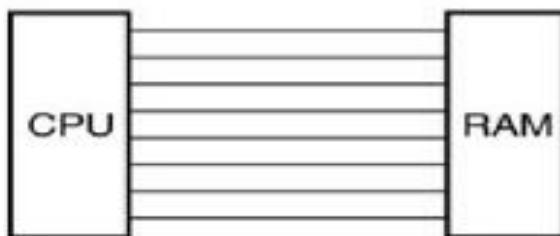
Countermeasure: Bus Scrambling

- **Data bus scrambling is used to confuse attackers**
 - Order of the data bus is changed to make it difficult to observe bus signals

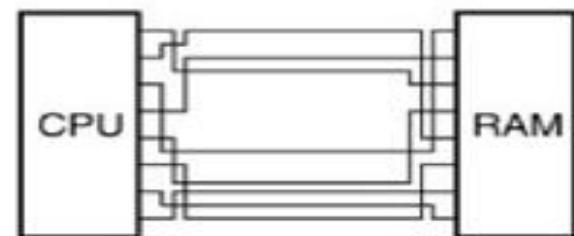


Bus scrambling

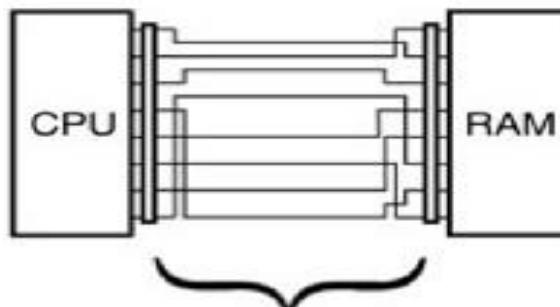
data bus with conventional chip layout



data bus with static scrambling

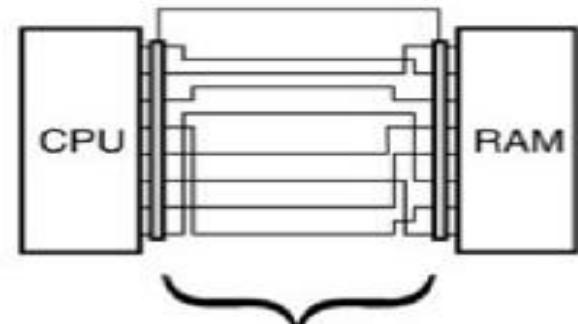


data bus with chip-specific scrambling



different for each
microcontroller

data bus with session-specific scrambling



different for each session
or portion of a session

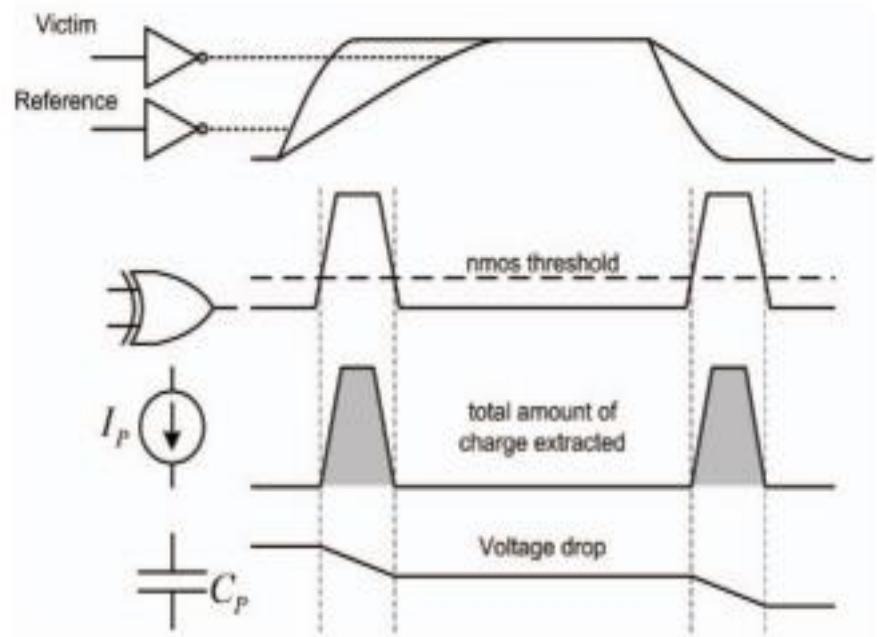
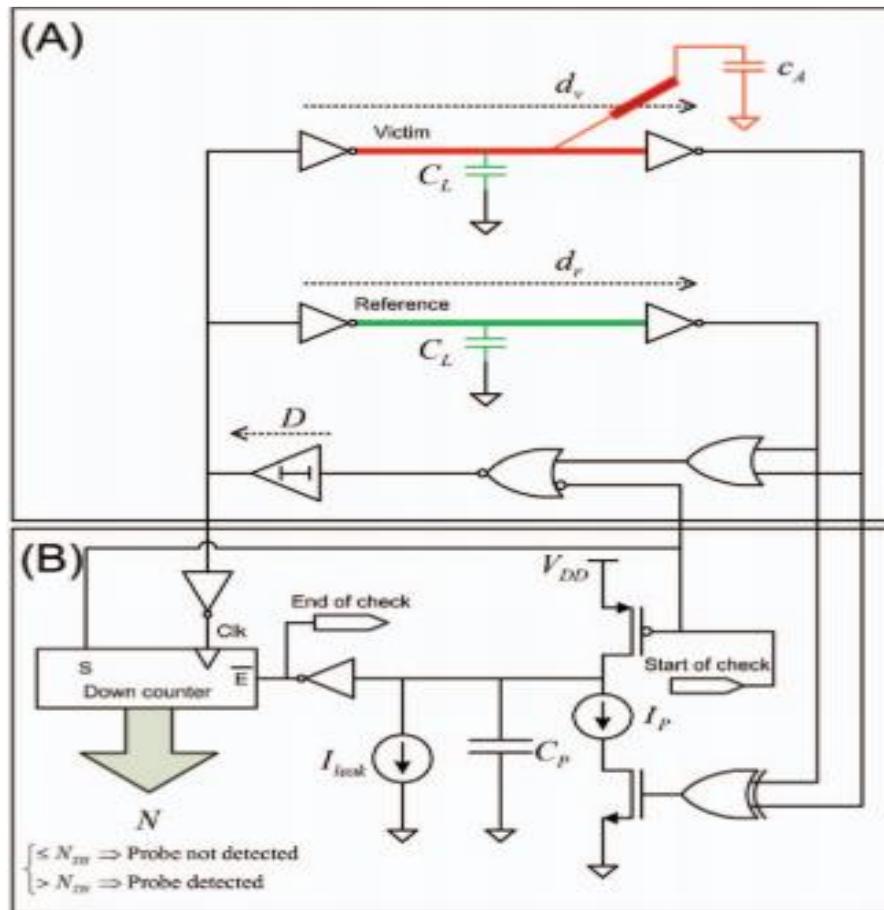
Countermeasures: Sensors

- Different kind of sensors can be used to detect attack attempt
 - Voltage and frequency sensors for glitching attacks
 - Light sensor can be helpful against decapsulation of the device
- Special purpose sensors can be created to detect probing
 - Ring oscillator based detector (Probing Attempt Detector)

Sensors: Probe Attempt Detector (PAD)

- Exploits the fact that probing will change the capacitance in the bus line.
 - Place ring oscillators on the bus lines
 - When the probe touches the one or more bus lines, frequency of the ring oscillator changes n Because of the added capacitance
- PAD observes the bus lines continuously, when they have significant difference, it sets a flag that there is a probing attempt on one of the lines

Sensors: Probe Attempt Detector (PAD)



ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 16: Physically Unclonable Functions

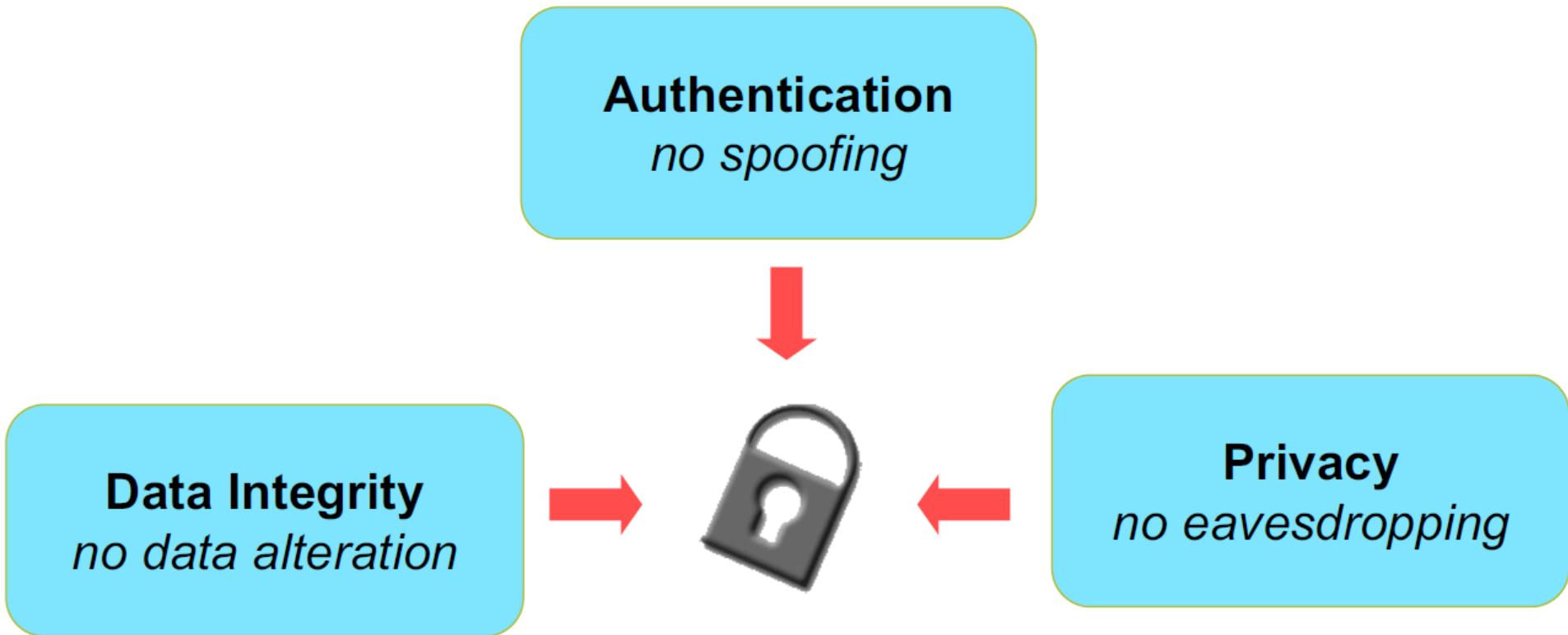
04/03/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Slides are adapted from Mark Tehranipoor, U. of Florida

What do we want to achieve?



Attacks



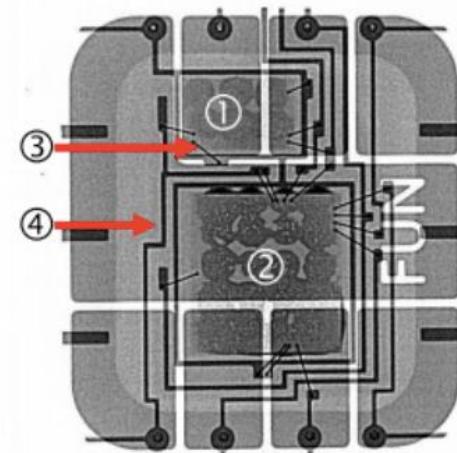
- Software-only protection is not enough. Non-volatile memory technologies are vulnerable to invasive attack as secrets always exist in digital form

Threat Model

■ Attacker goals

- To get the crypto keys stored in RAM or ROM
- To learn the secret crypto algorithm used
- To obtain other information stored into the chip (e.g.PINs)
- To modify information on the card (e.g. calling card balance)

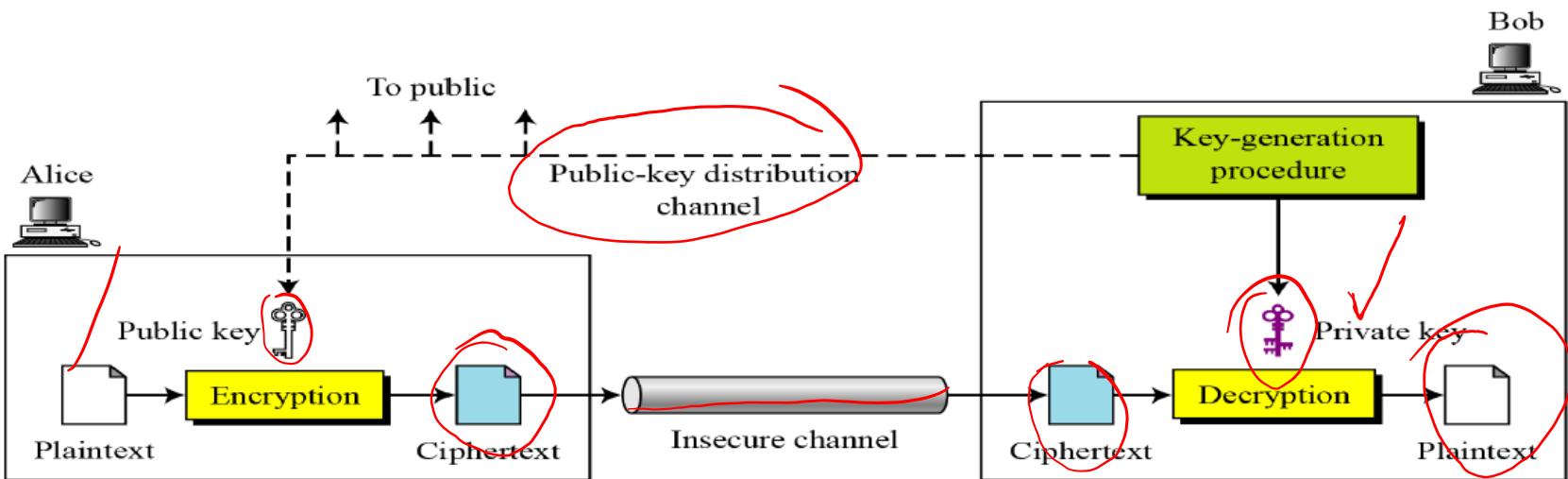
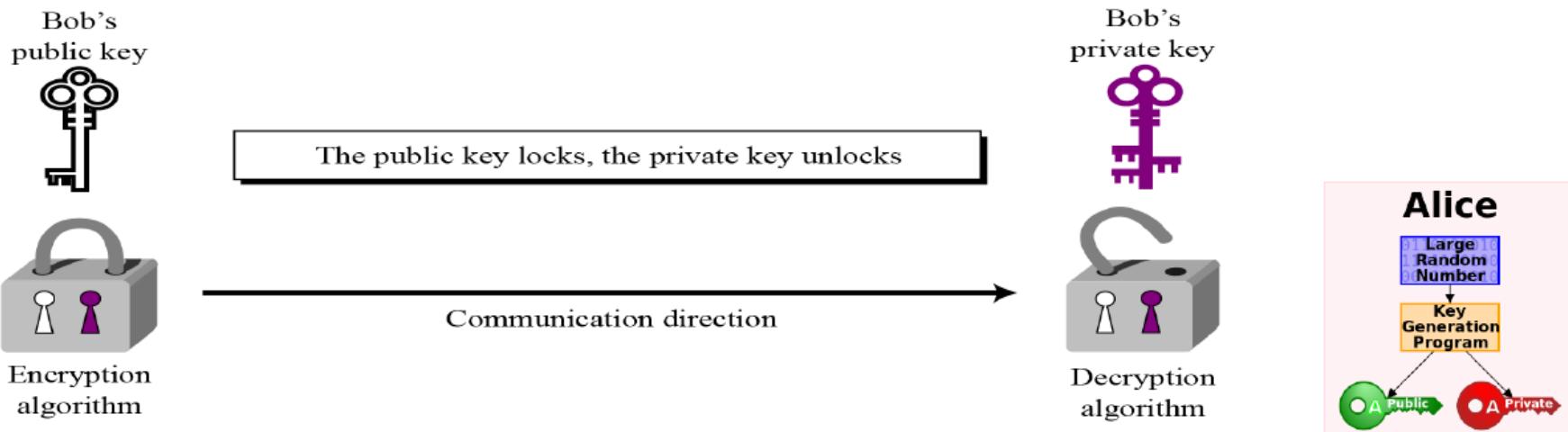
Over \$680,000 stolen via a clever man-in-the-middle attack on chip cards in 2011



Basic Terminologies

- Keys are rules used in algorithms to convert a document into a secret document
- Keys are of two types:
 - Symmetric
 - Asymmetric
- A key is **symmetric** if the same key is used both for encryption and decryption
- A key is **asymmetric** if different keys are used for encryption and decryption

Asymmetric Security



Security

- Asymmetry b/w the information (secret)
- **One-way functions**
 - Easy to evaluate in one direction but hard to reverse in the other
 - E.g., multiplying large prime number as opposed to factoring them
- **One-way hash functions**
 - Maps a variable length input to a fixed length output
 - **Avalanche property:** changing one bit in the input alters nearly half of the output bits
 - Pre-image resistant, collision resistant
 - Usage: digital signature, secured password storage, file identification, and message authentication code

Challenges of algorithmic (mathematical) one-way functions

- **Technological**

- Massive number of parallel devices broke DES
- Reverse-engineering of secure processors

- **Fundamental**

- There is no proof that attacks do not exist
- E.g., quantum computers could factor two large prime numbers in polynomial time

- **Practical**

- Embedded systems applications

Solution: POWF

- Use the chaotic physical structures that are hard to model instead of mathematical one-way functions!
- Physical One Way Functions (POWF)
 - Inexpensive to fabricate
 - Prohibitively difficult to duplicate
 - No compact mathematical representation
 - Intrinsically tamper-resistant

IBM 4758

- **Problem:**

- Storing digital information in a device in a way that is resistant to physical attack is difficult and expensive.

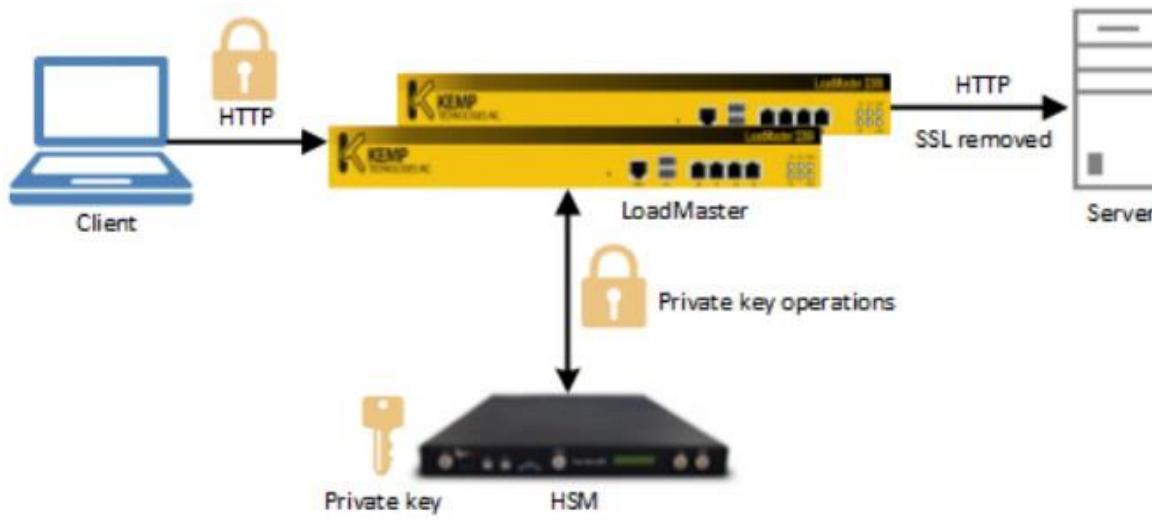
- **IBM 4758**

- Tamper-proof package containing a secure processor which has a secret key and memory
- Tens of sensors, resistance, temperature, voltage, etc.
- Continually battery-powered
- ~ \$3000 for a 99 MHz processor and 128MB of memory



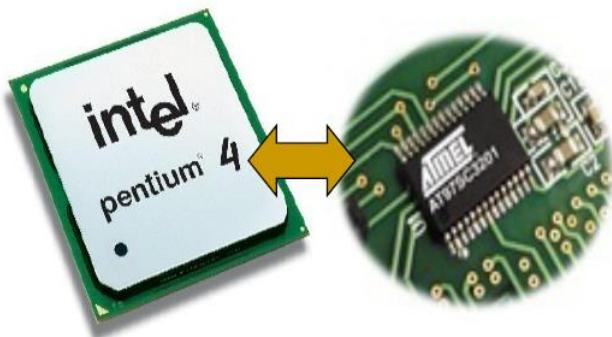
HSM

- A **hardware security module (HSM)** is a physical computing device that safeguards and manages digital keys for strong authentication and provides crypto processing. These modules traditionally come in the form of a plug-in card or an external device that attaches directly to a computer or network server. - Wikipedia



TPM

- A **Trusted Platform Module (TPM)** is a specialized chip on an endpoint device that stores RSA encryption keys specific to the host system for hardware authentication. Each TPM chip contains an RSA key pair called the Endorsement Key (EK). -- Wikipedia

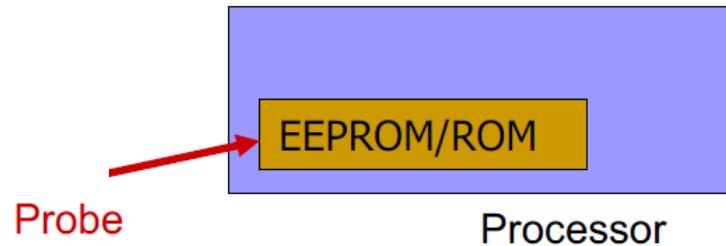


A separate chip (TPM) for security functions

Decrypted “secondary” keys can be read out from the bus

Problem

- Storing digital information in a device in a way that is resistant to physical attacks is difficult and expensive.



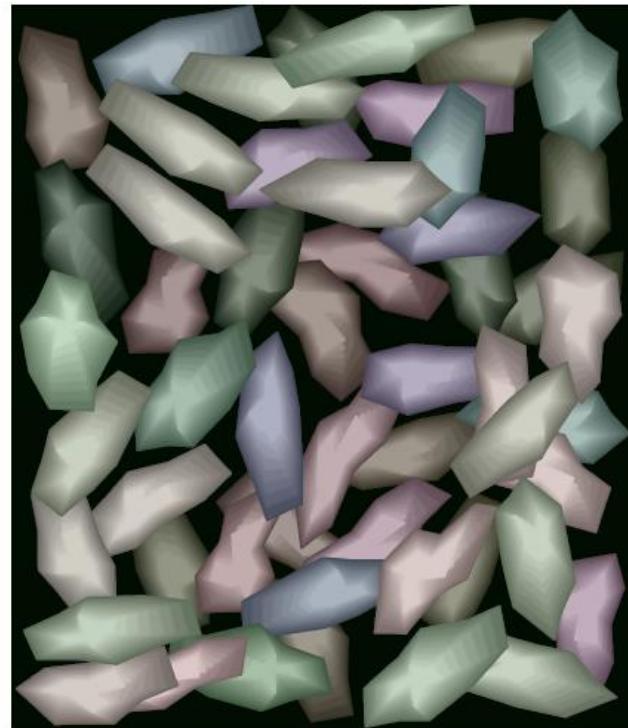
- Adversaries can physically extract secret keys from EEPROM while processor is off
 - Trusted party must embed and test secret keys in a secure location
 - EEPROM adds additional complexity to manufacturing

Feature: Process Variation

- Do we expect process variation (length, widths, oxide thickness) in circuit and system?
 - Impact circuit performance
 - Functional failure
 - Major obstacle to the continued scaling of integrated-circuit technology in the sub-45 nm regime
- Process variations can be turned into a feature rather than a problem?
 - Each IC has unique properties

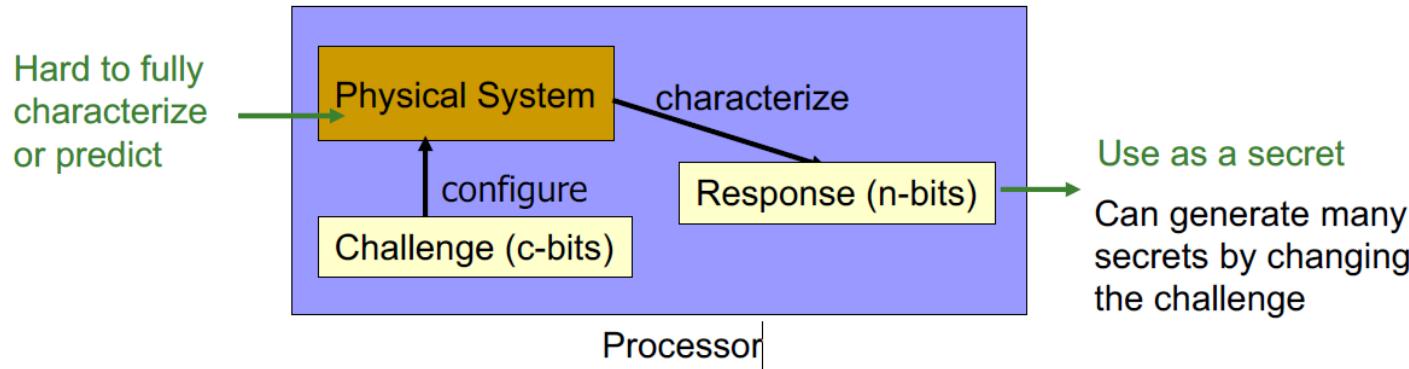
Solution

- **Extract key information from a complex physical system**



Physical Unclonable/Random Functions (PUFs)

- Generate keys from a **complex physical system**



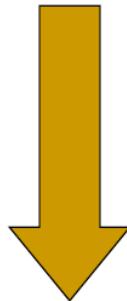
- Security Advantage
 - Keys are generated on demand -> No non-volatile secrets
 - No need to program the secret
 - Can generate multiple master keys
- What can be **hard to predict**, but **easy to measure**?

Definition

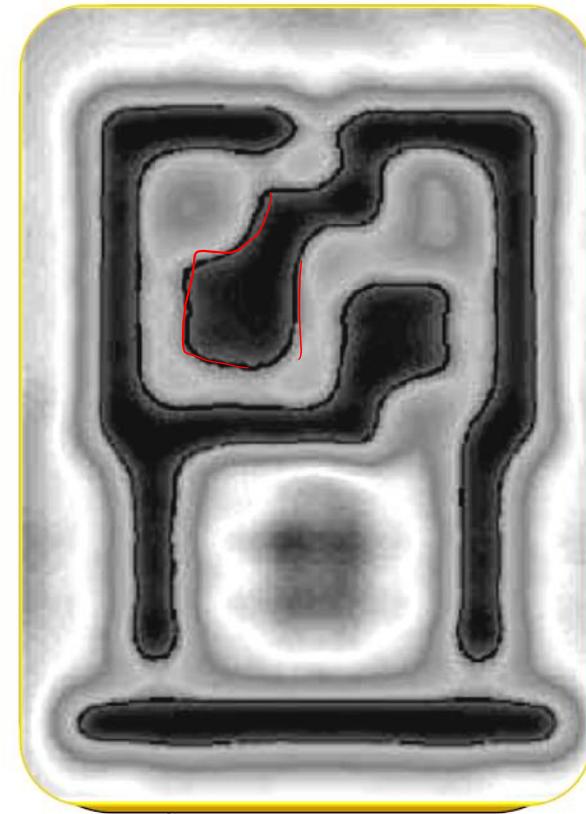
- Physical Random Function or **Physical Unclonable Function (PUF)** is a function that is:
 - Based on a physical system
 - Easy to evaluate (using the physical system)
 - Its output looks like a random function
 - Unpredictable even for an attacker with physical access

WYSINWYG

Sub-Wavelength WYSINWYG



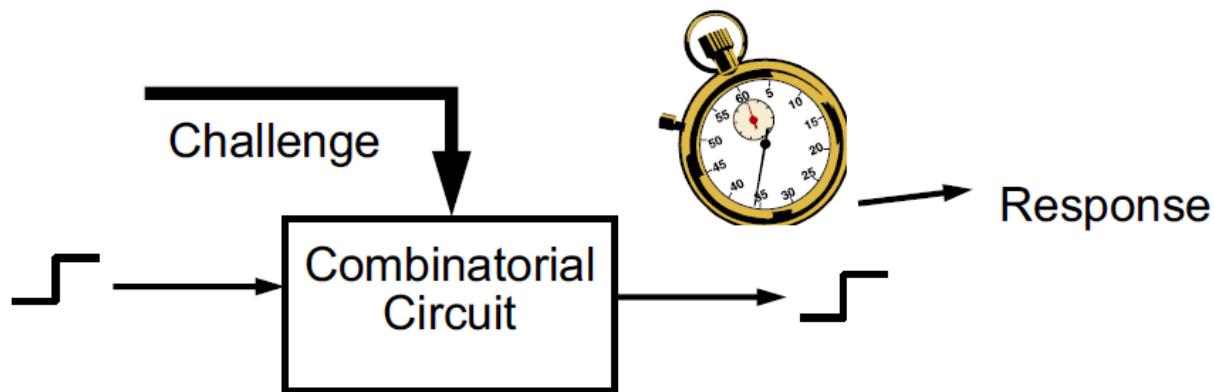
What You See Is Not What You Get



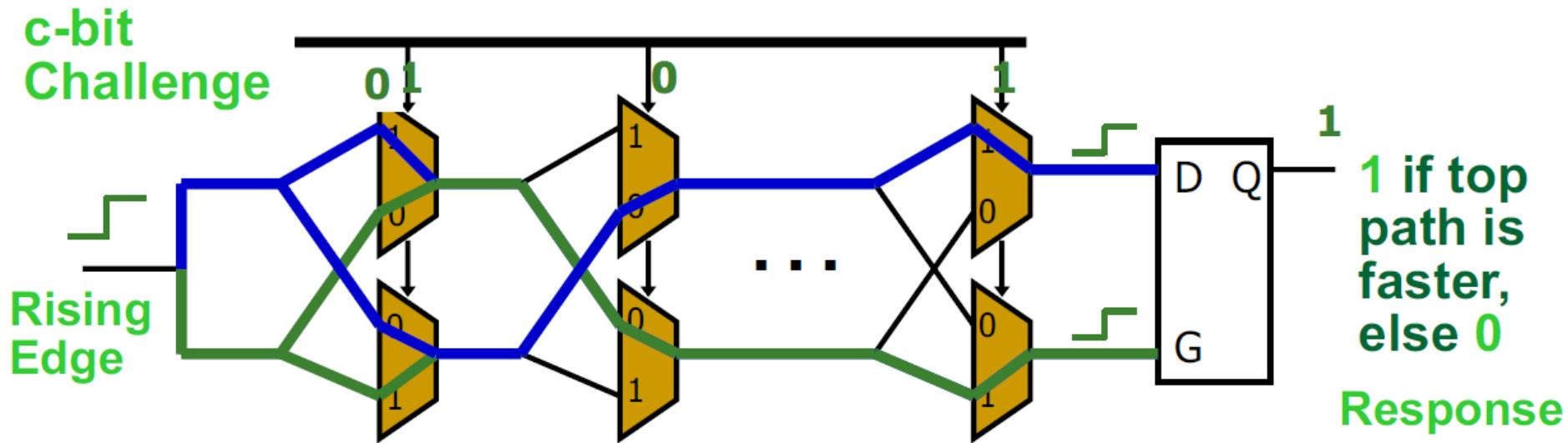
- Process variations
- No two transistors have the same parameters

Silicon PUF – Proof of Concept

- Because of process variations, **no two Integrated Circuits are identical**
- Experiments in which *identical circuits with identical layouts* were placed on different FPGAs show that path delays vary enough across ICs to use them for identification.



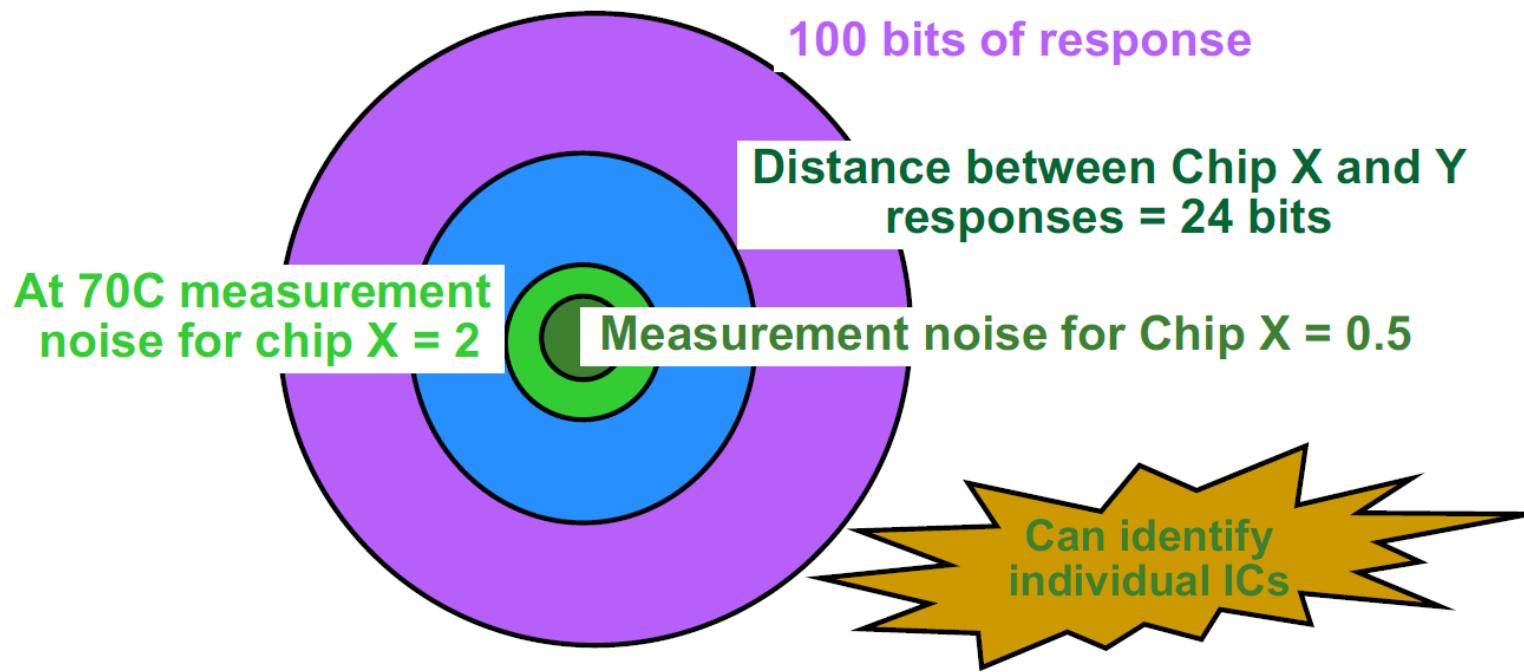
A candidate: Silicon PUF (Arbiter)



- Compare two paths with an identical delay in design
 - Random process variation determines which path is faster
 - An arbiter outputs 1-bit digital response
- **Path delays in an IC are statistically distributed due to random manufacturing variations**

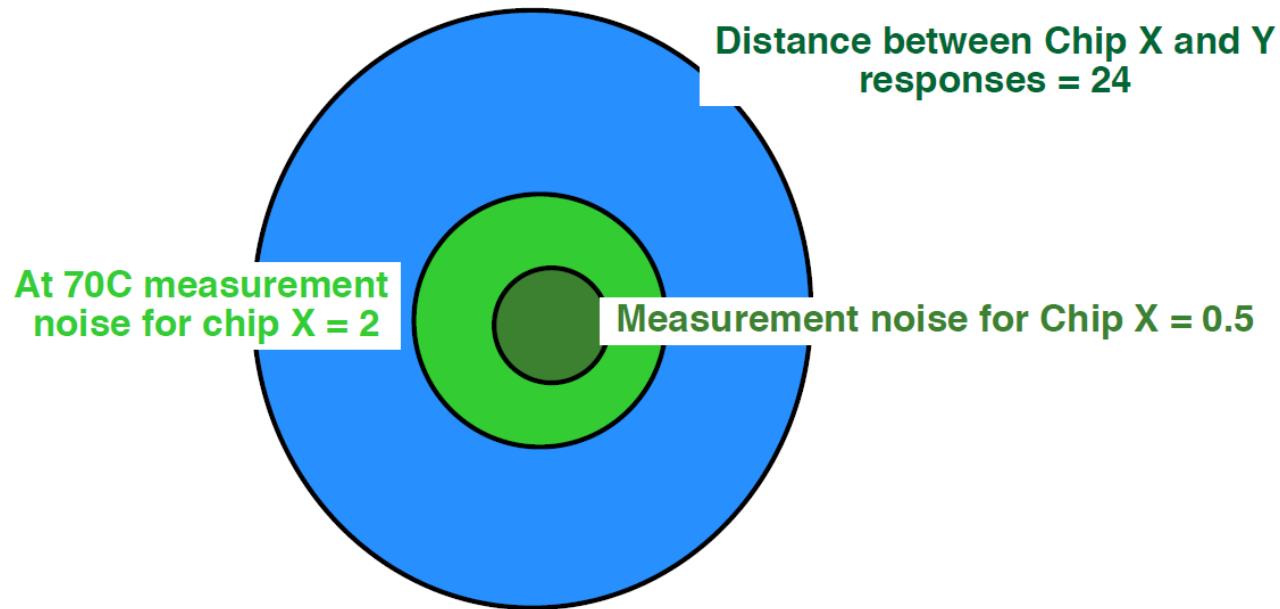
Experiments

- Fabricated candidate PUF on multiple ICs, 0.18um TSMC
- Apply 100 random challenges and observe responses



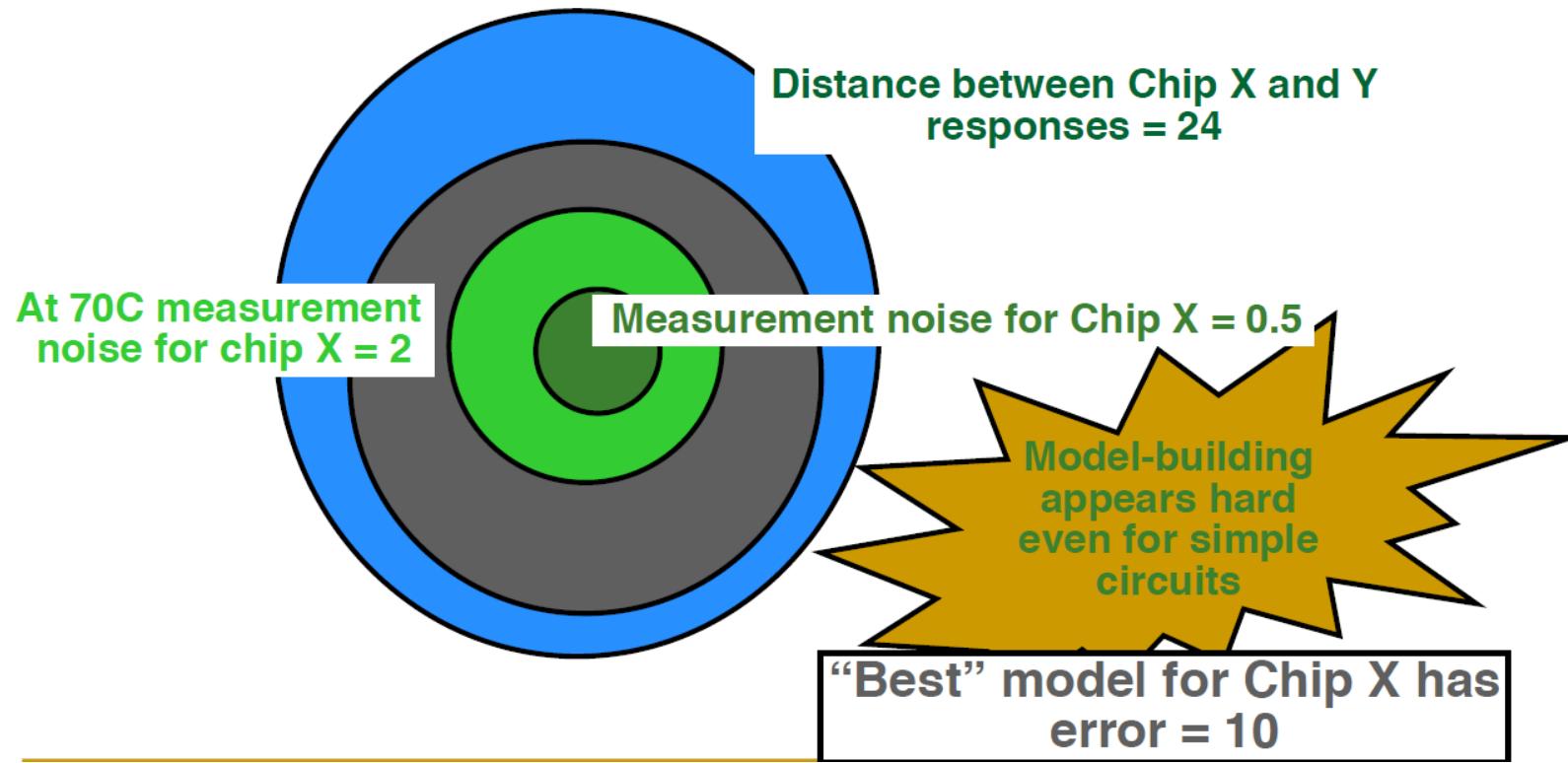
Measurement Attacks and Software

- Can an adversary create a *software clone* of a given PUF chip?



Measurement Attacks and Software Attacks

- Can an adversary create a *software clone* of a given PUF chip?

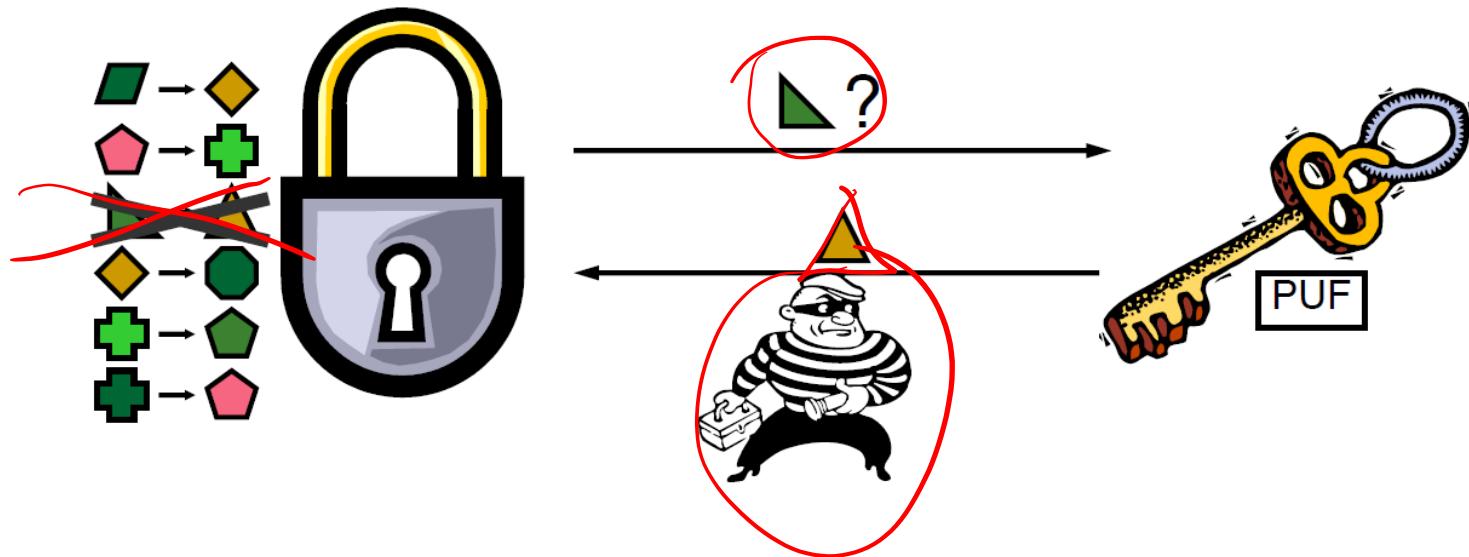


Physical Attacks

- Make PUF delays depend on overlaid metal layers and package
- Invasive attack (e.g., package removal) changes PUF delays and destroys PUF
- Non-invasive attacks are still possible
 - To find wire delays one needs to find precise relative timing of transient signals as opposed to looking for 0's and 1's
 - Wire delay is not a number but a function of challenge bits and adjacent wire voltages and capacitances

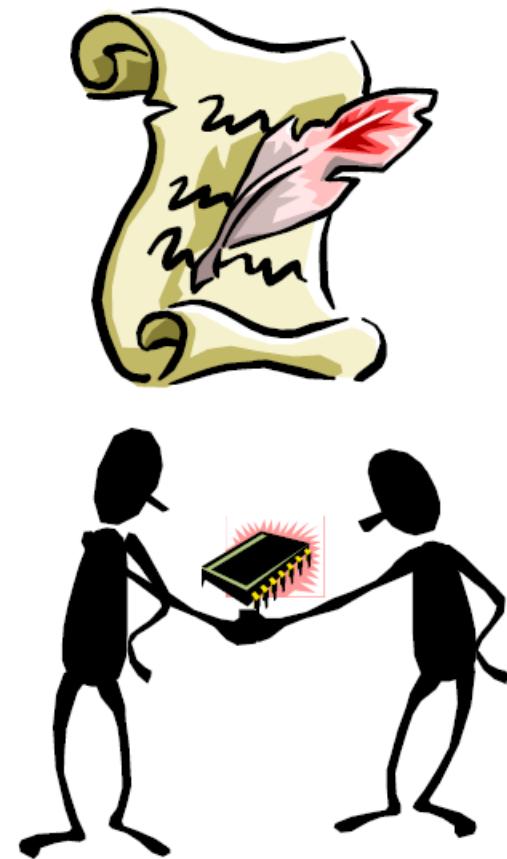
Using a PUF as an Unclonable Key

- A Silicon PUF can be used as an unclonable key.
 - The lock has a database of challenge-response pairs.
 - To open the lock, the key has to show that it knows the response to one or more challenges.



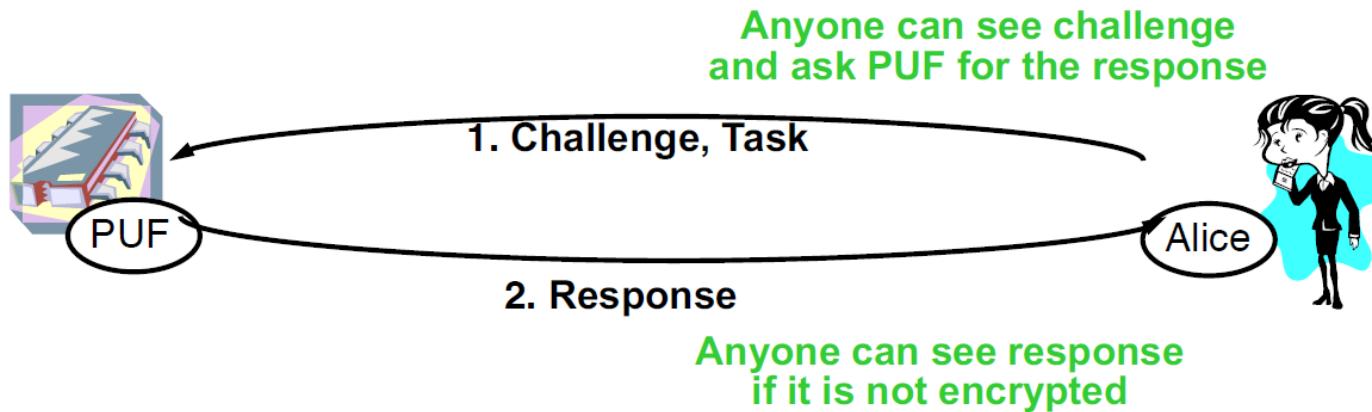
Applications

- **Anonymous Computation**
- Alice wants to run computations on Bob's computer, and wants to make sure that she is getting correct results. A certificate is returned with her results to show that they were correctly executed.
- **Software Licensing**
- Alice wants to sell Bob a program which will only run on Bob's chip (identified by a PUF). The program is copy-protected so it will not run on any other chip.
- We can enable the above applications by trusting only a single-chip processor that contains a silicon PUF



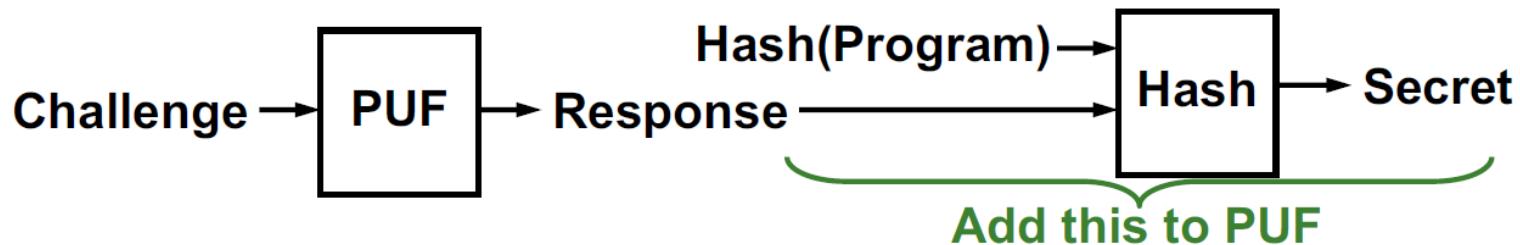
Sharing a Secret with a Silicon PUF

- Suppose Alice wishes to share a secret with the silicon PUF
- She has a challenge response pair that no one else knows, which can authenticate the PUF
- She asks the PUF for the response to a challenge



Restricting Access to the PUF

- To prevent the attack, the man in the middle must be prevented from finding out the response.
 - Alice's program must be able to establish a shared secret with the PUF, the attacker's program must not be able to get the secret.
- > **Combine response with hash of program.**
- The PUF can only be accessed via the **GetSecret** function:



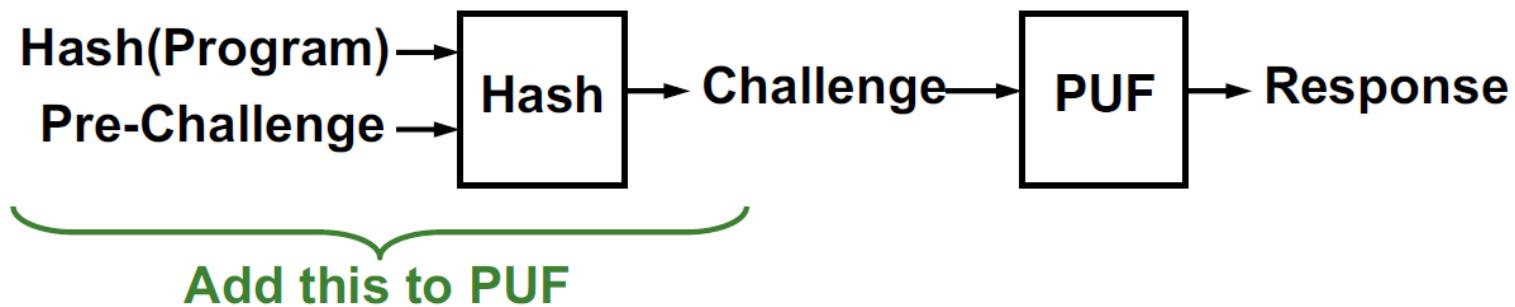
Getting a Challenge-Response Pair

- Now Alice **can** use a Challenge-Response pair to generate a shared **secret** with the PUF equipped device.
- But Alice **can't** get a Challenge-Response pair in the first place since the PUF **never** releases responses directly.

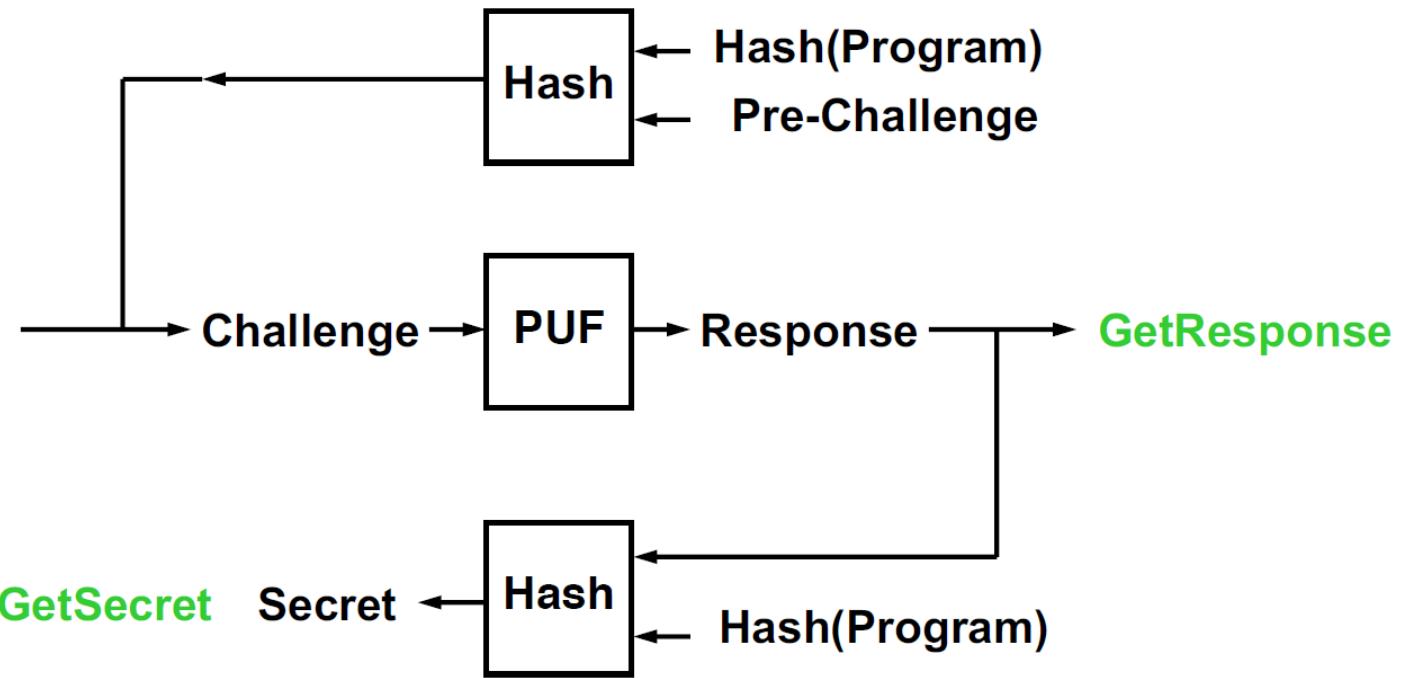
An extra function that can return responses is needed.

Getting a Challenge-Response Pair 2

- Let Alice use a **Pre-Challenge**.
- Use program hash to prevent eavesdroppers from using the pre-challenge.
- The PUF has a **GetResponse** function

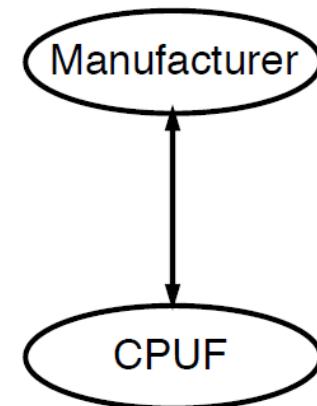


Controlled PUF Implementation



Challenge-Response Pair Management: Bootstrapping

- When a Controlled PUF (CPUF) has just been produced, the manufacturer wants to generate a challenge-response pair.
 - Manufacturer provides **Pre-challenge** and **Program**.
 - CPUF produces **Response**.
 - Manufacturer gets **Challenge** by computing $\text{Hash}(\text{Hash}(\text{Program}), \text{PreChallenge})$.
 - Manufacturer has $(\text{Challenge}, \text{Response})$ pair where **Challenge**, **Program**, and **Hash(Program)** are public, but **Response** is not known to anyone since **Pre-challenge** is thrown away



Software Licensing

Program (Ecode, Challenge)
Secret = GetSecret(Challenge)
Code = Decrypt(Ecode, Secret)
Run Code

$\left. \begin{array}{l} \text{Secret} = \text{GetSecret}(\text{Challenge}) \\ \text{Code} = \text{Decrypt}(\text{Ecode}, \text{Secret}) \end{array} \right\} \text{Hash}(\text{Program})$

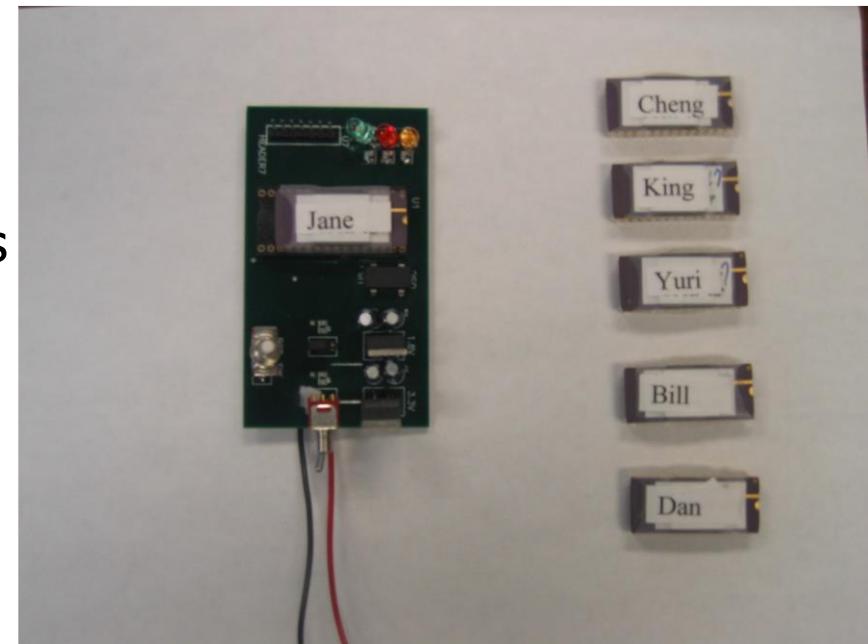
- **Ecode** has been encrypted with **Secret** by Manufacturer
- **Secret** is known to the manufacturer because he knows Response to Challenge and can compute **Secret = Hash(Hash(Program), Response)**
- Adversary cannot determine **Secret** because he does not know **Response** or **Pre-Challenge**
- If adversary tries a different program, a different secret will be generated because **Hash(Program)** is different

Summary

- PUFs provide secret “key” and CPUFs enable sharing a secret with a hardware device
- CPUFs are not susceptible to model-building attack if we assume physical attacks cannot discover the PUF response
 - Control protects PUF by obfuscating response, and PUF protects the control from attacks by “covering up” the control logic
 - Shared secrets are volatile

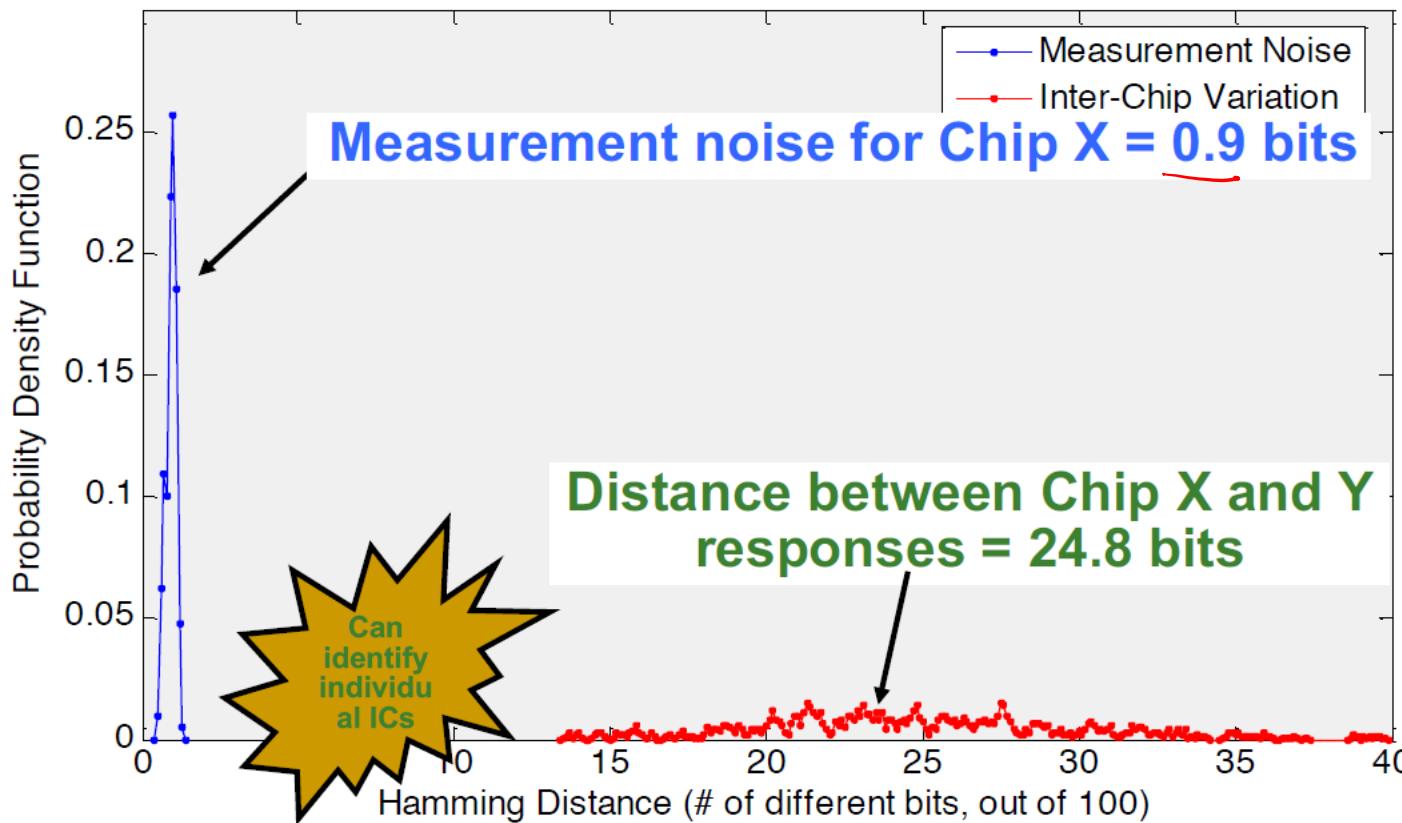
PUF Experiments

- Fabricated 200 “identical” chips with PUFs in TSMC 0.18 μ on 5 different wafer runs
- **Security**
 - What is the probability that a challenge produces different responses on two different PUFs?
- **Reliability**
 - What is the probability that a PUF output for a challenge changes with temperature?
 - With voltage variation?



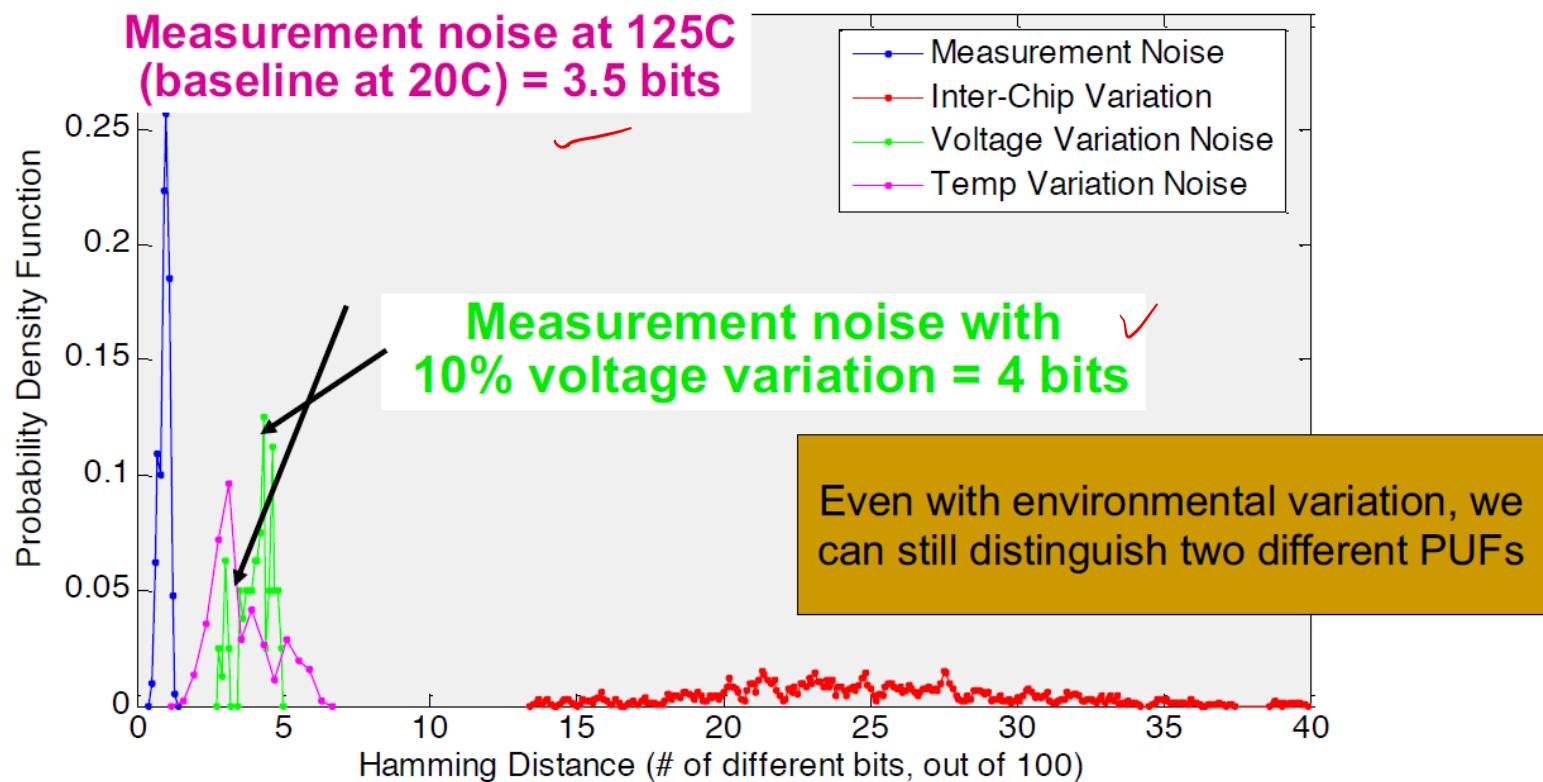
Inter-Chip Variation

- Apply random challenges and observe 100 response bits



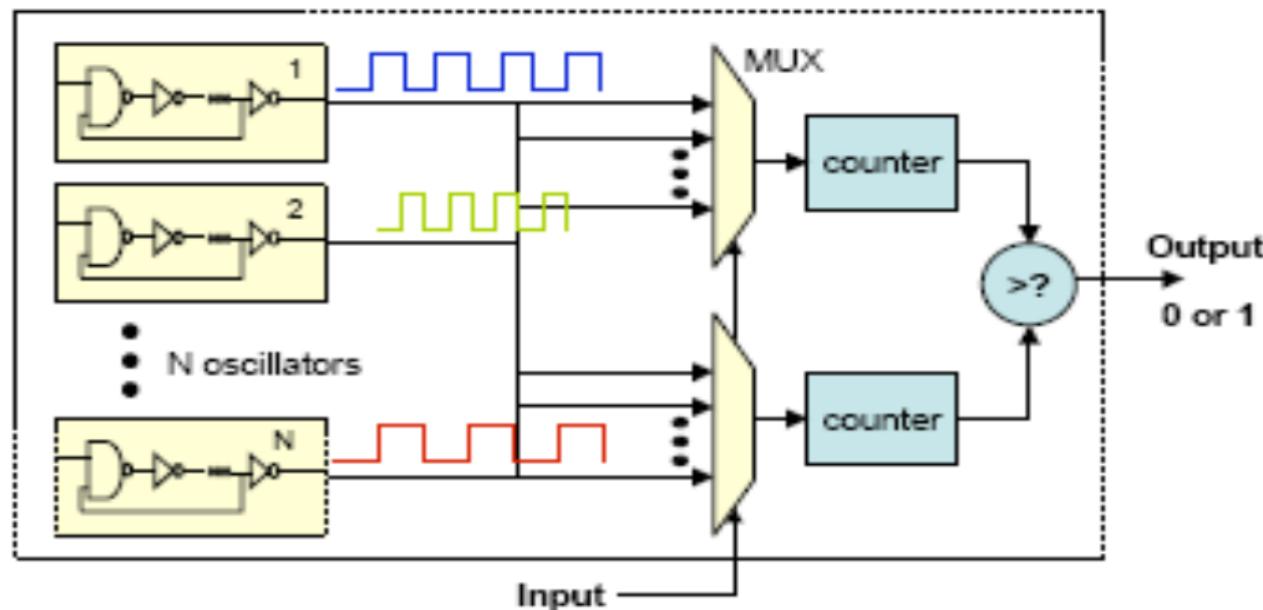
Environmental Variations

- What happens if we change voltage and temperature?



Ring-Oscillator (RO) PUF

- The structure relies on delay loops and counters instead of MUX and arbiters
- Better results on FPGA – more stable

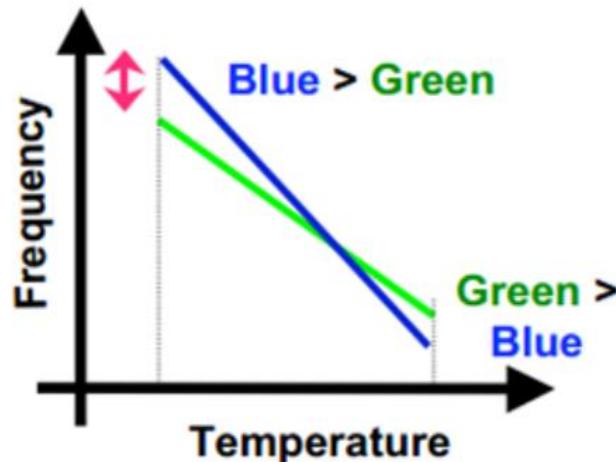


RO PUFs (cont'd)

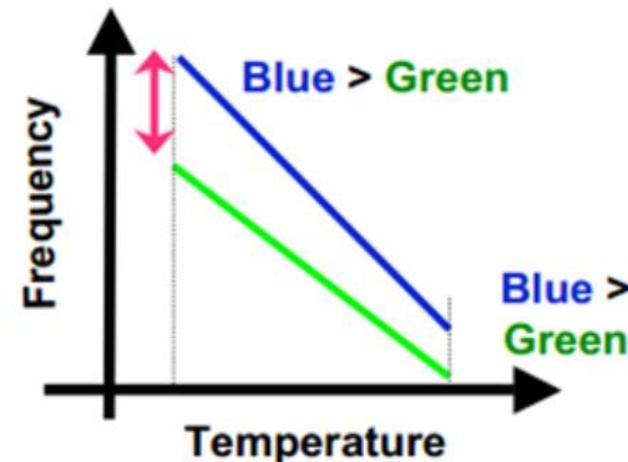
- Easy to duplicate a ring oscillator and make sure the oscillators are identical
 - Much easier than ensuring the racing paths with equal path segments
- How many bits can we generate from the scheme in the previous page?
 - There are $N(N-1)/2$ distinct pairs, but the entropy is significantly smaller: $\log_2(N!)$
 - E.g., 35 ROs can produce 133 bits, 128 ROs can produce 716, and 1024 ROs can produce 8769

Reliability Enhancement

- Environmental changes have a large impact on the freq. (and even relative ones)



(a) Frequencies are close



(b) Frequencies are far apart

RO PUFs

- ROs whose frequencies are far are more stable than the ones with closer frequencies
 - Possible advantage: do not use all pairs, but only the stable ones
 - It is easy to watch the distance in the counter and pick the very different ones.
 - Can be done during enrollment
- RO PUF allows an easier implementation for both ASICs and FPGAs.
- The **Arbiter** PUF is appropriate for resource constrained platforms such as RFIDs and the **RO PUF** is better for use in FPGAs and in secure processor design.

Experiments with RO PUFs

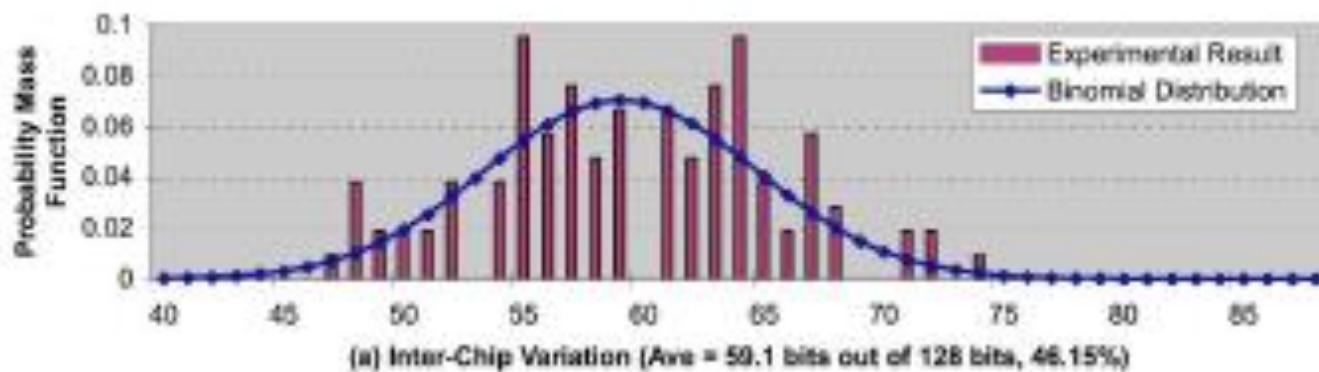
- Experiments done on 15 Xilinx Virtex4 LX25 FPGA (90nm)
- They placed 1024 ROs in each FPGA as a 16-by-64 array
- Each RO consisted of 5 INVs and 1 AND, implemented using look-up tables
- The goal is to know if the PUF outputs are unique (**for security**) and reproducible (**for reliability and security**)

Reliability and Security Metrics

- *Inter-chip variation:* How many PUF output bits are different between PUF A and PUF B? This is a measure of uniqueness. If the PUF produces uniformly distributed independent random bits, the inter-chip variation should be 50% on average.
- *Intra-chip (environmental) variation:* How many PUF output bits change when re-generated again from a single PUF with or without environmental changes? This indicates the *reproducibility* of the PUF outputs. Ideally, the intra-chip variation should be 0%.

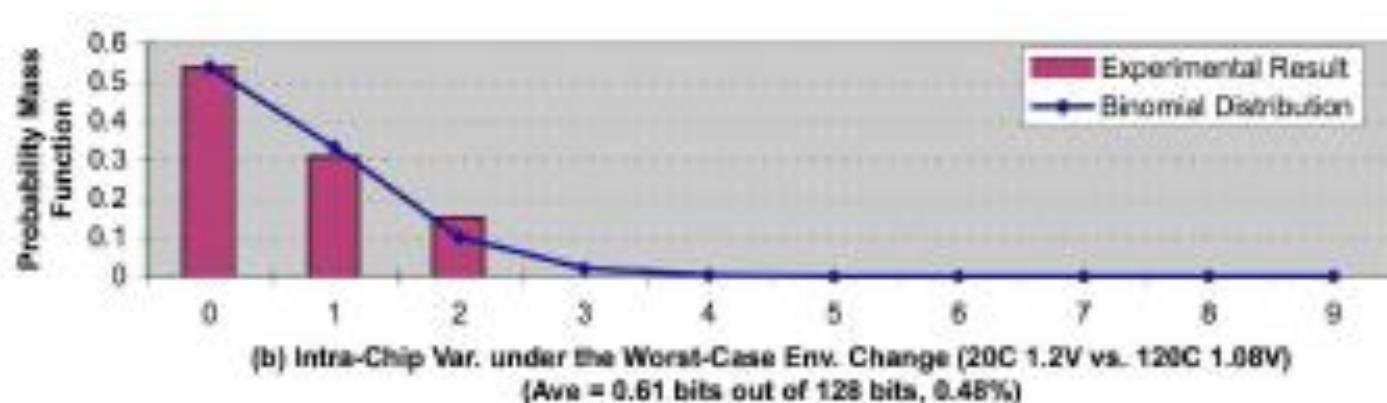
The Probability Distribution for Inter-Chip Variations

- 128 bits are produced from each PUF
- x-axis: number of PUF o/p bits different b/w two FPGAs; yaxis: probability
- Purple bars show the results from 105 pair-wise comparisons
- Blue lines show a binomial distribution with fitted parameters ($n=128$, $p = 0.4615$)
- Average inter-chip variations $0.4615 \sim 0.5$



The Probability Distribution for Intra-Chip Variations

- PUF responses are generated at two different conditions and compared
- Changing the temperature from 20°C to 120°C and the core voltage from 1.2 to 1.08 altered the PUF o/p by ~0.6 bits (0.47%)
- Intra-chip variations is much lower than inter-chip – the PUF o/p did not change from small to moderate



Configurable Ring Oscillator

- The pair which has the maximum difference in frequency in CRO is selected

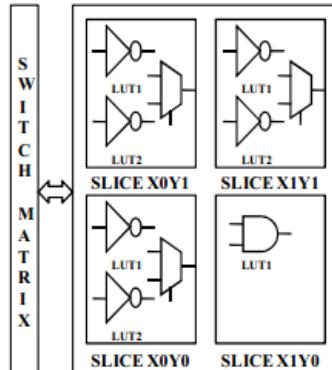
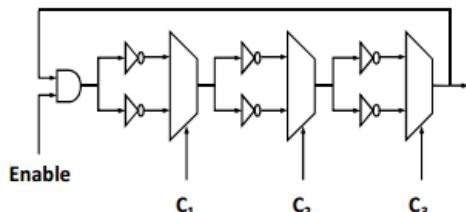
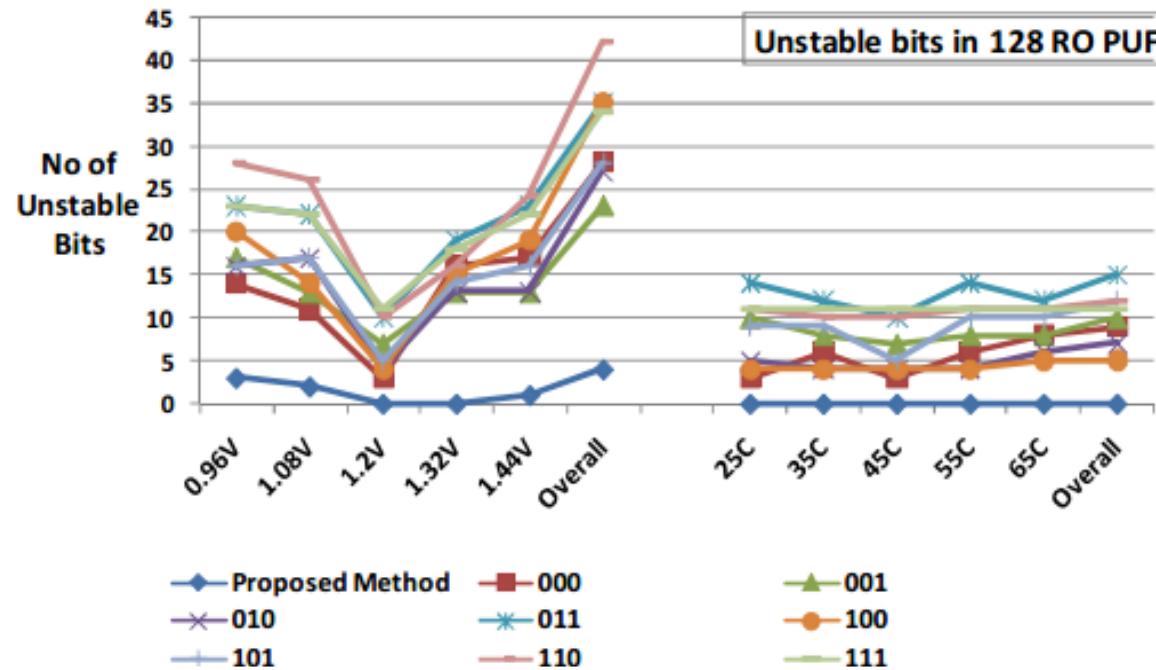


Table 1. Frequency differences in a configurable RO pair

$c_1c_2c_3$	Frequency of ROs in CLB i	Frequency of ROs in CLB j	Δf
000	f_0	f'_0	$ f_0 - f'_0 $
001	f_1	f'_1	$ f_1 - f'_1 $
010	f_2	f'_2	$ f_2 - f'_2 $
011	f_3	f'_3	$ f_3 - f'_3 $
100	f_4	f'_4	$ f_4 - f'_4 $
101	f_5	f'_5	$ f_5 - f'_5 $
110	f_6	f'_6	$ f_6 - f'_6 $
111	f_7	f'_7	$ f_7 - f'_7 $

Configurable Ring Oscillator



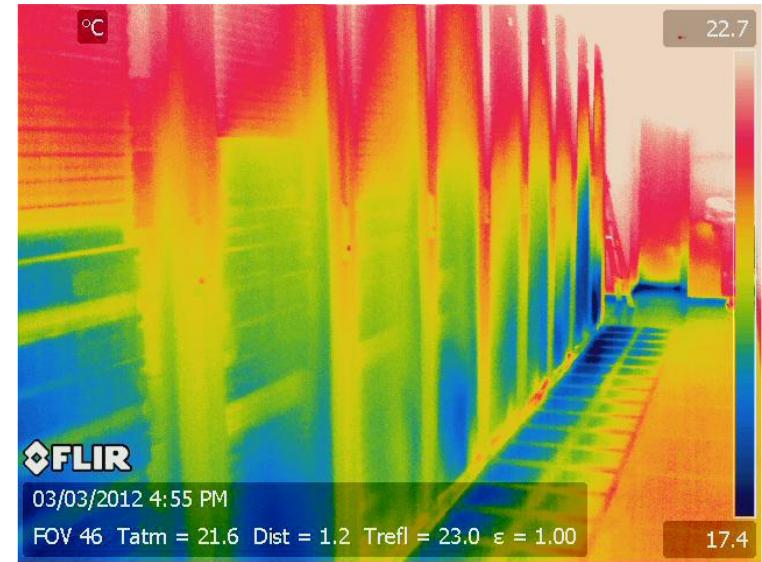
- Higher difference in frequency will ensure higher reliability
- Redundancy

More on PUF Applications

- Securely Sensing without Key

Motivation: Securely Sensing without Key

- Physical features should be closely monitored
 - The temperature of data center, Facebook, Google, Amazon, etc.



Motivation: Securely Sensing without Key

- Physical features should be closely monitored
 - The temperature of data center, Facebook, Google, Amazon, etc
- Some physical features might be meaningful for security/privacy
 - Relative location between bank cards and automated teller machines (ATMs), card (not) present withdrawal



Motivation: Securely Sensing without Key

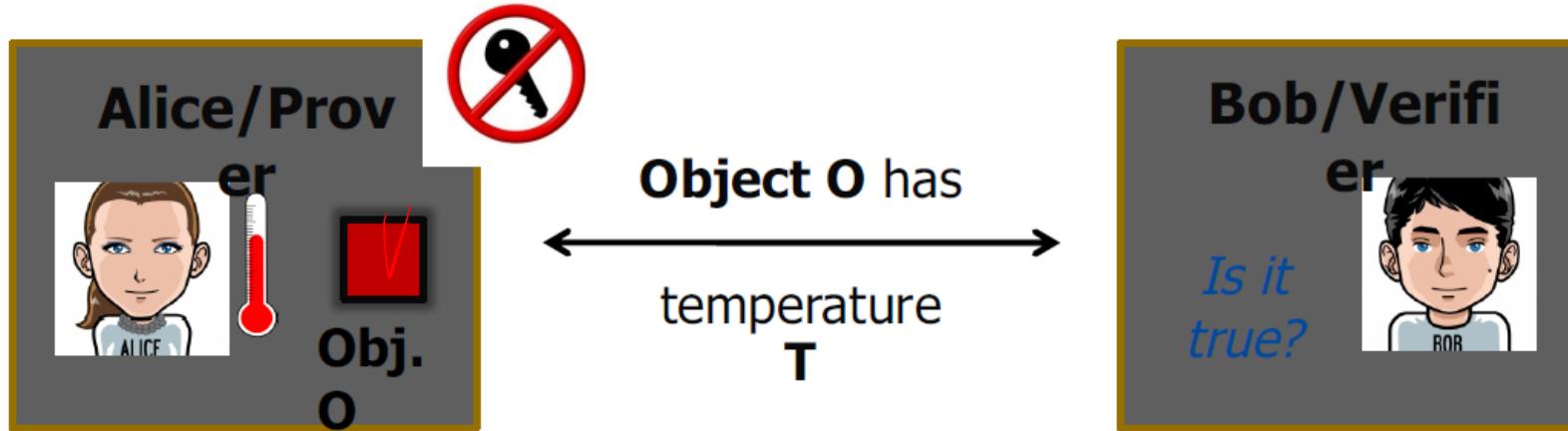
- Some physical features are hard to sense but favoring many applications
 - Digital rights management, physically/irreversibly canceling membership?
- **Secure sensors are needed:**
 - Operation safety, secrecy of sensitive data

Key Is a Target

- Modern security protocols are commonly based on secret keys.
- A robust key enhances the robustness of security systems, but also announces itself as an interested target for attackers.

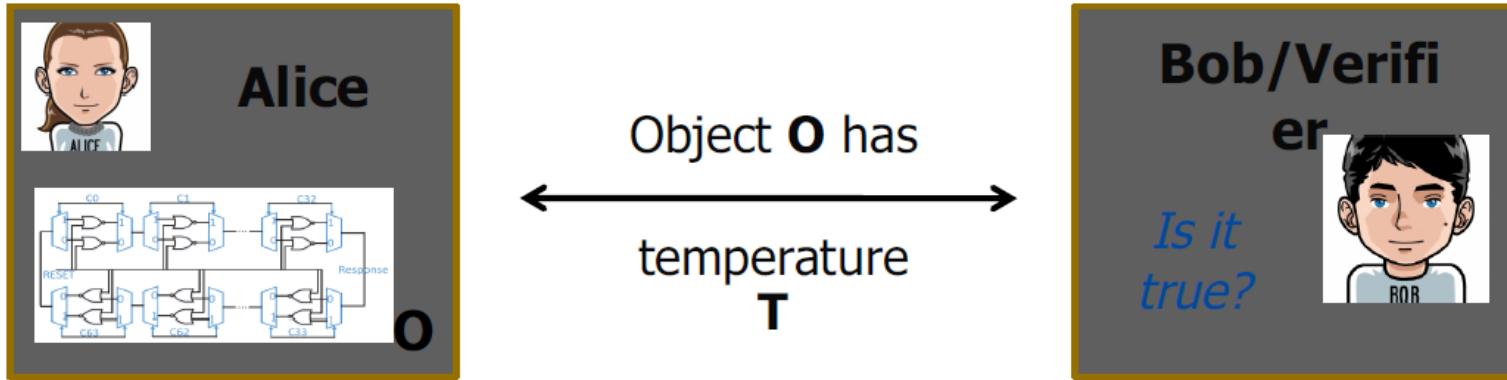


Virtual Proofs of Temperature (Or: Keyless Temperature Sensors)



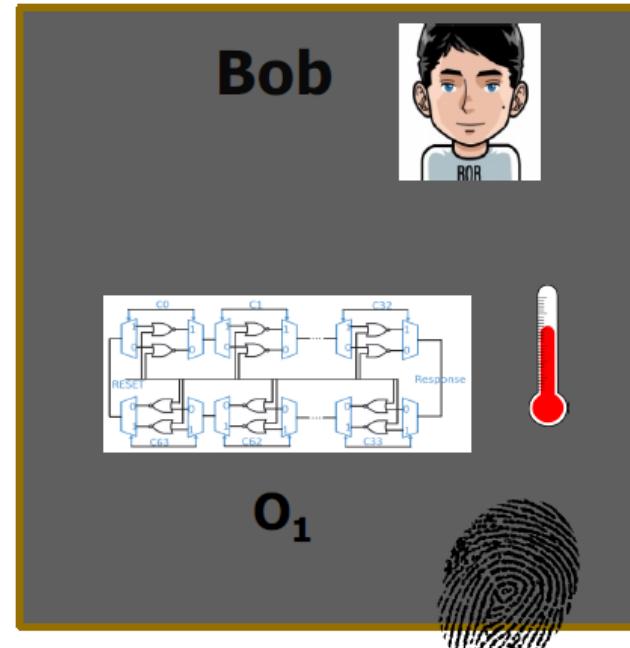
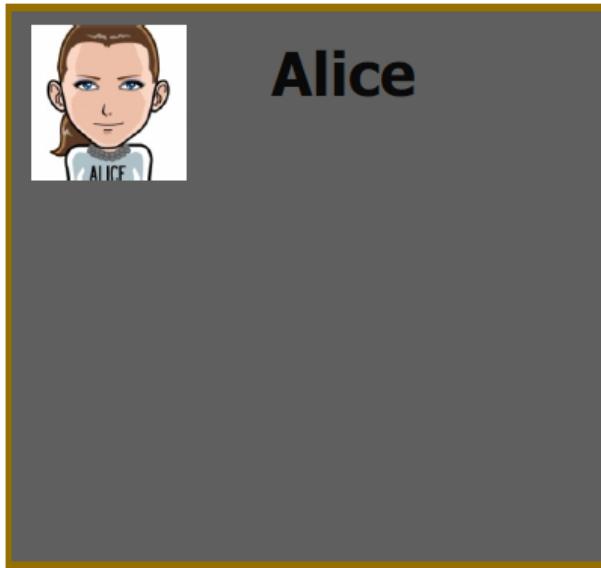
- Alice wants to prove to Bob that
 - a certain **object O**
 - is at temperature **T**
 - at the time of the execution of the VP
- **Classical approach:** Secure sensors with key known to Bob...
 - But, of course, we want no keys...

VPs of Temperature via Noise Sensitivity



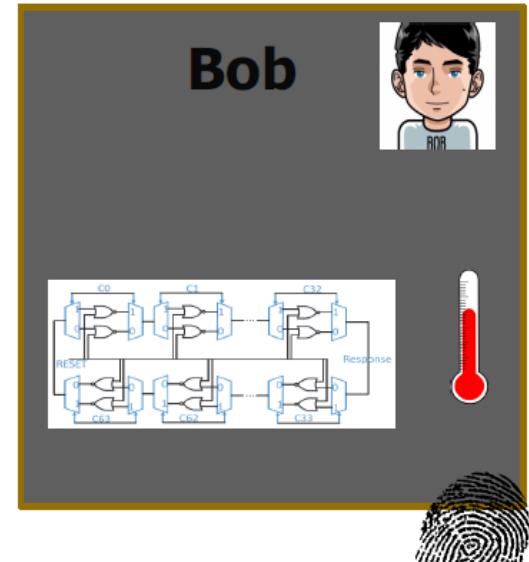
- **Key Idea:** Use a ***temperature-sensitive Strong PUF*** as **Object O**
 - ie., an integrated circuit (**IC**) / **PUF** whose output **intentionally** depends upon the ambient temperature **T**
 - Output of the IC/PUF to Bob's random challenges proves temperature

VP of Temperature $T_i \in \{T_1, \dots, T_k\}$



- For each temperature $T_i \in \{T_1, \dots, T_k\}$:
 - **Bob** puts the objects O_1 at temperature T_i
 - Chooses n random **challenges** C_1^i, \dots, C_n^i
 - Measures the n resulting **responses**: r_1^i, \dots, r_n^i
 - **Bob** creates&stores private list $\text{List}(T_i) = (C_1^i, r_1^i), \dots, (C_n^i, r_n^i)$

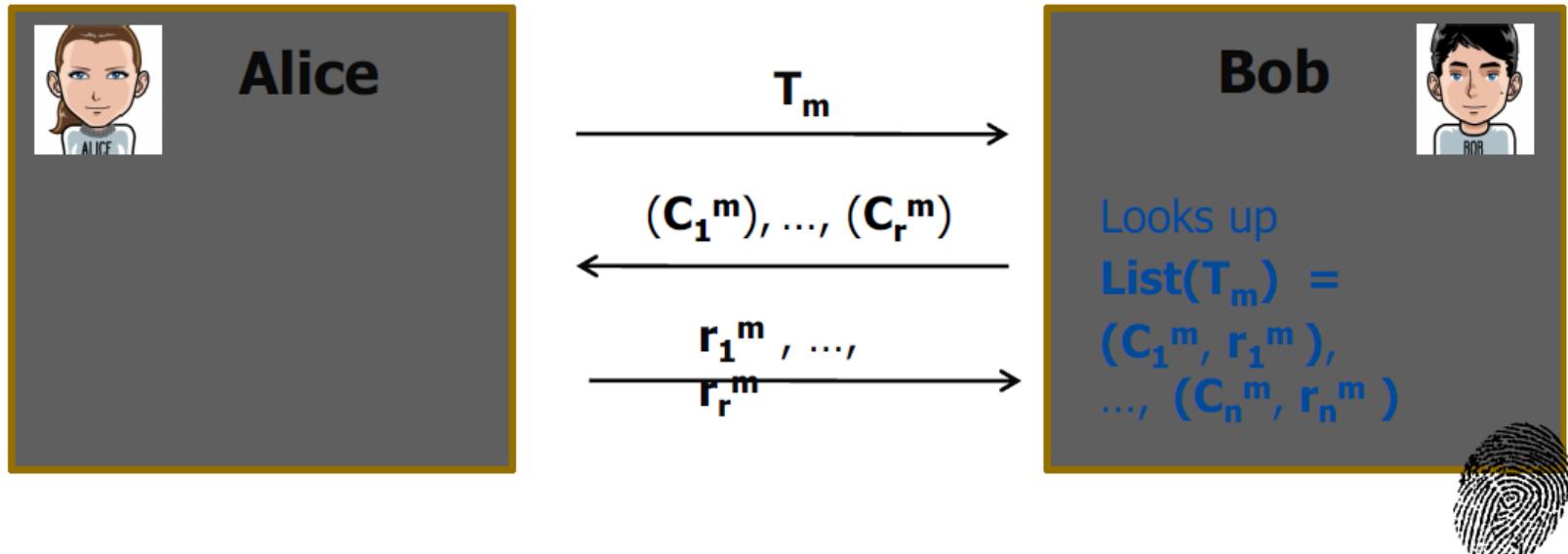
VP of Temperature $T_i \in \{T_1, \dots, T_k\}$



- Set-up phase closes, and the objects are transferred to **Alice**
- Some time may pass... (time for Alice to attack the system...)
- **VP starts!!**



VP of Temperature $T_i \in \{T_1, \dots, T_k\}$



- If pre-recorded values r_i^m in the **List(T_m)** **match** the values that **Alice** sent, then **Bob** will **accept** the **VP**



ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 17: Side-Channel Attacks

04/05/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

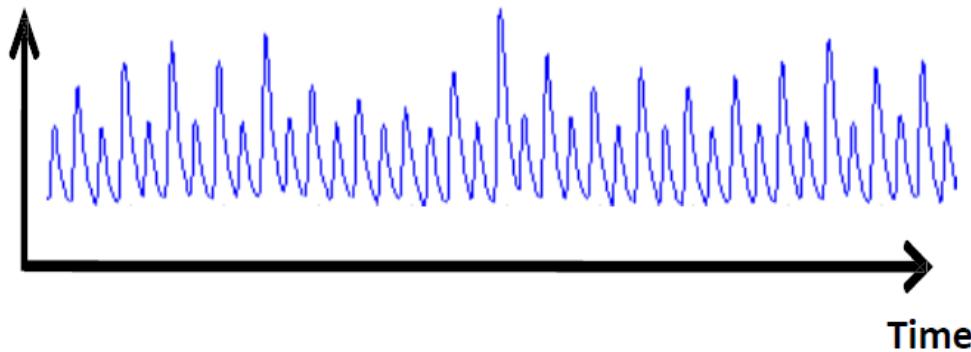
Slides are adapted from Mark Tehranipoor, U. of Florida

What Really is Side-Channel Attack

- Side-Channel attacks aim at **side-channel inputs and outputs**, bypassing the theoretical strength of cryptographic algorithms
 - In computer security, a **side-channel attack** is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs). Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.
- Five commonly exploited side-channel emissions:
 - Power Consumption
 - Electro-Magnetic
 - Optical
 - Timing and Delay
 - Acoustic

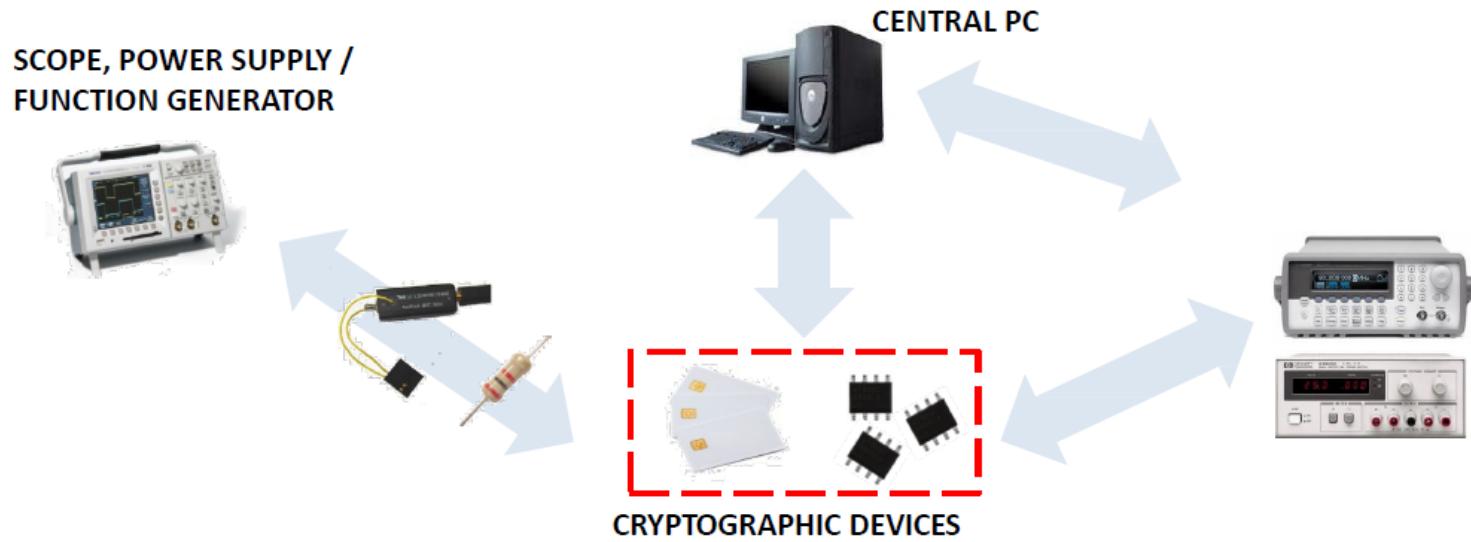
Measuring Power Consumption

- Not average power over time, not peak power
- Instantaneous power over time
 - Trace or curve, many samples



Measuring Power Consumption

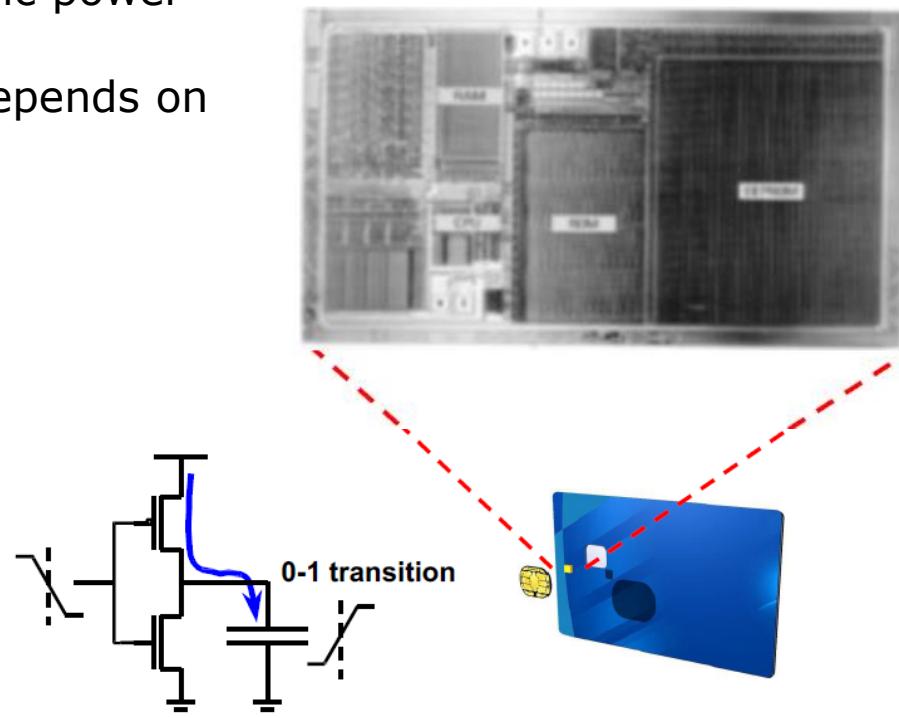
- Typical (automated) measurement setup



Measuring Power Consumption

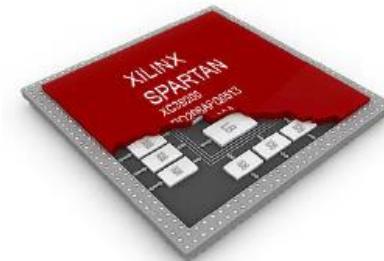
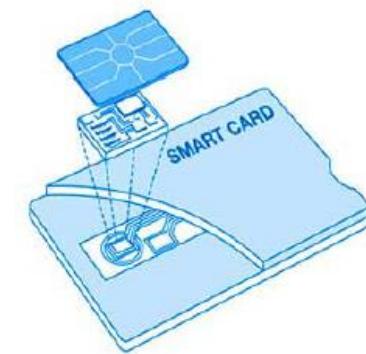
- **Logic:** constant supply voltage, supply current varies
- **Predominant technology:** CMOS
 - Low static power consumption
 - Relatively high dynamic power consumption
 - Power consumption depends on input
- **CMOS inverter:**

Input	Output	Current
$0 \rightarrow 0$	$1 \rightarrow 1$	Low
$0 \rightarrow 1$	$1 \rightarrow 0$	Discharge
$1 \rightarrow 0$	$0 \rightarrow 1$	Charge
$1 \rightarrow 1$	$0 \rightarrow 0$	Low



Hardware Targets

- Two common victims of hardware cryptanalysis are **smart cards** and **FPGAs**
 - Attacks on smart cards are applicable to any general purpose processor with a fixed bus architecture.
 - Attacks on FPGAs are also reported. FPGAs represent application specific devices with parallel computing opportunities.



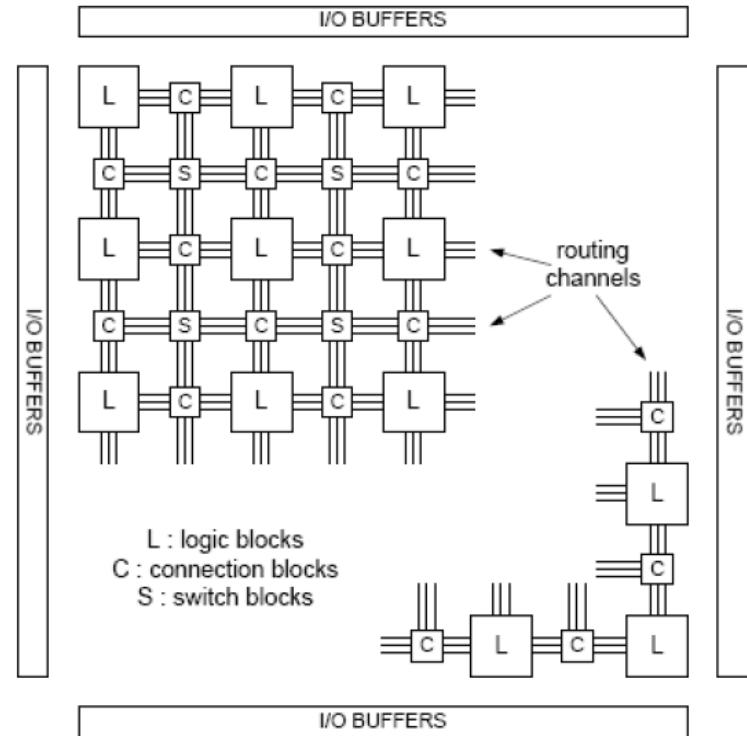
Smart Cards

- Smart cards have a small processor (8bit in general) with ROM, EEPROM and a small RAM
 - **Eight wires** connect the processor to the outside world
 - **Power supply**: There is no internal battery
 - **Clock**: There is no internal clock
 - Typically equipped with a **shield** that destroys the chip if a tampering happens



FPGAs

- FPGAs allow parallel computing
- Multiple programmable configuration bits



Attack Model / Assumptions

- Consider a device capable of implementing the cryptographic function
- The key is usually stored in the device and protected
- Modern cryptography is based on **Kerckhoffs's assumption** -> all of the data required to operate chip is entirely hidden in the key
- ***Attacker only needs to extract the key***

Attack Phases

- Such attacks are usually composed of two phases:
 - **Interaction phase:** interact with the hardware system under attack and obtain the physical characteristics of the device
 - **Analysis phase:** analyze the gathered information to recover the key

Principle of divide-and-conquer attack

- The divide-and-conquer (D&C) attack attempts at recovering the key by parts
- The idea is that **an observed characteristic can be correlated with a partial key**
 - The partial key should be small enough to enable exhaustive search
- Once a partial key is validated, the process is **repeated** for finding the remaining keys
- D&C attacks may be iterative or independent

Attack Classification

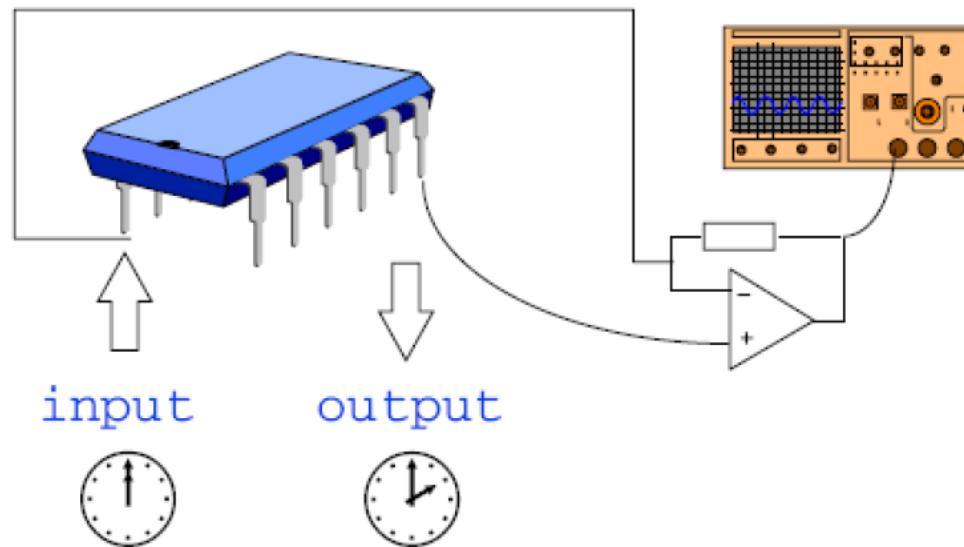
- **Invasive** vs. **noninvasive** attacks
- **Active** vs. **passive** attacks
 - Active attacks exploit side-channel inputs
 - Passive attacks exploit side-channel outputs

Attack Classification

- **Simple vs. differential** attacks
 - Simple side-channel attacks directly map the results from a small number of traces of the side-channel to the *operation* of device under attack
 - Differential side-channel attacks exploit the correlation between the *data values* being processed and the side-channel *leakage*

Power Attacks

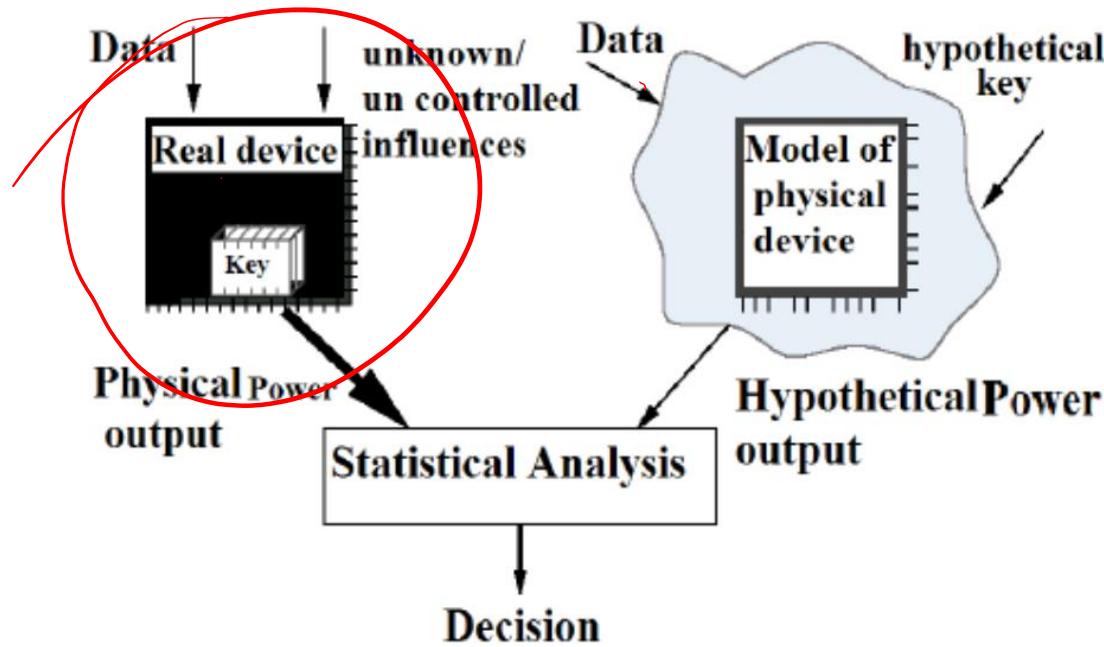
- Measure the circuit's processing time and current consumption to infer what is going on inside it.



Measuring Phase

- The task is usually straightforward
 - Easy for smart cards: the energy is provided by the terminal and the current can be read
- Relatively inexpensive (<\$1000) equipment can digitally sample voltage differences at high rates (1GHz++) with less than 1% error
- Device's power consumption depends on many things, including its structure and data being processed

Power Attacks



Simple Power Analysis (SPA)

- Originally proposed by Paul Kocher, 1996
- Monitor the device's power consumption to deduce information about data and operation

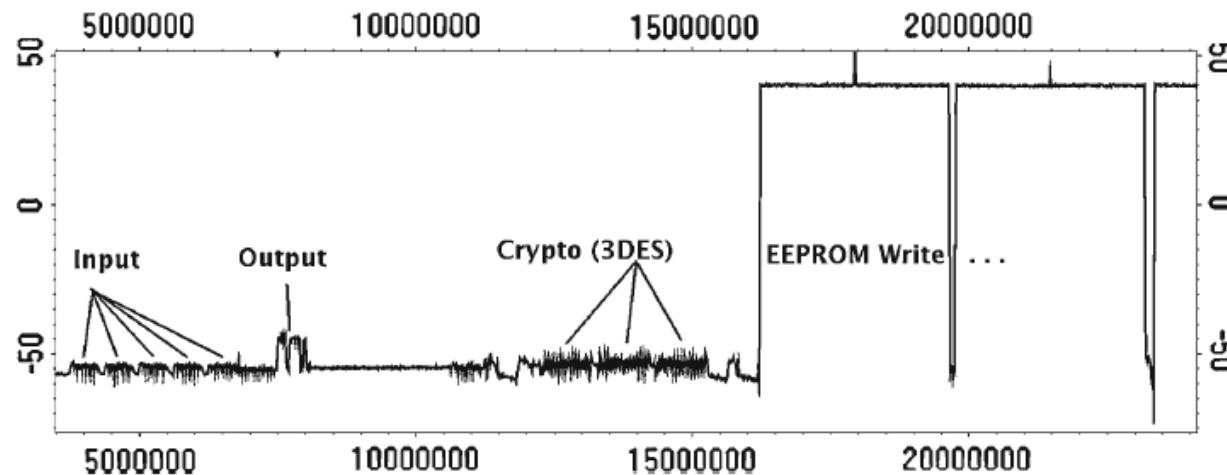
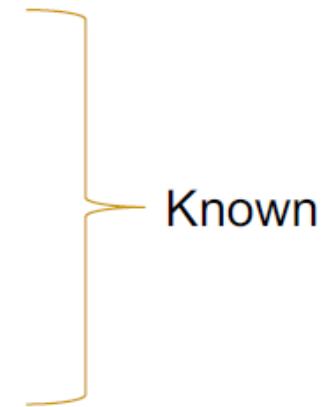


Fig. 10 Power trace from a smart card that is performing a 3DES-based PRNG operation

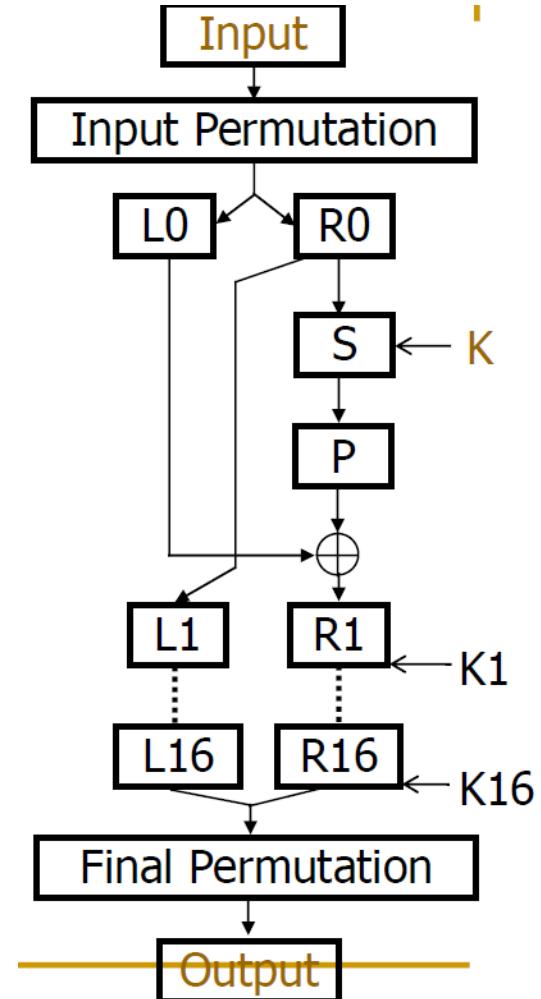
SPA

- Example: SPA on DES – smart cards
 - The internal structure is shown on the next slide
- Summary of DES – a block cipher
 - a product cipher
 - 16 rounds iterations
 - substitutions (for confusion)
 - permutations (for diffusion)
- Each round has a *round key*
 - Generated from the user-supplied key

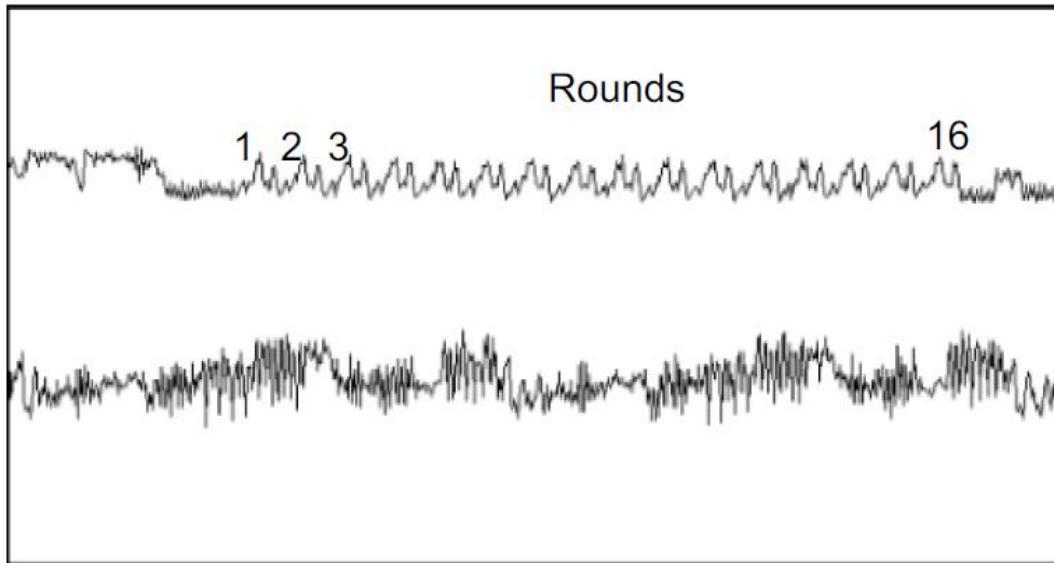


DES (DATA Encryption Standard) Basic structure

- Input: 64 bits (a block)
- L_i/R_i - left/right half (32 bits) of the input block for iteration i - subject to substitution S and permutation P
- K - user-supplied key
- K_i - round key:
 - 56 bits used + 8 unused (unused for encryption but often used for error checking)
- Output: 64 bits (a block)
- Note: R_i becomes L_{i+1}
- All basic op's are simple logical ops
 - Left shift / XOR



SPA on DES



- The upper trace – entire encryption, including the initial phase, 16 DES rounds, and the final permutation
- The lower trace – detailed view of the second and third rounds
- **The power trace can reveal the instruction sequence**

SPA

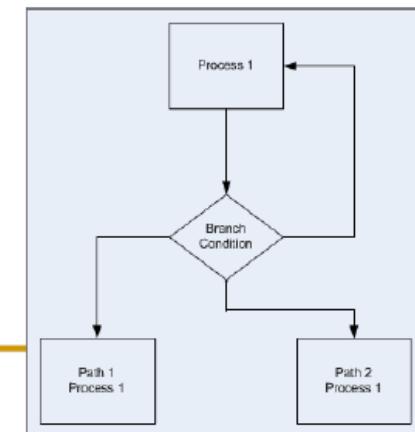
- SPA can be used to break cryptographic implementations (execution path, instruction, key change, etc.)
 - **DES key schedule:** Involves rotating 28-bit key registers
 - **DES permutation:** involves conditional branching
 - **Comparison:** Involves string and memory comparison operations performing a conditional branch when a mismatch is found
 - **Multipliers:** Involves modular multiplication – The leakage function depends on the multiplier design but strongly correlated to operand values and Hamming weights
 - **Exponentiators:** Involves squaring operation and multiplication
- SPA Countermeasure:
 - Avoid procedures that use secret intermediates or keys for conditional branching operation

SPA

- The DES structure and 16 rounds are known
- **Instruction flow depends on data -> power signature**
- Example: Modular exponentiation in DES is often implemented by square and multiply algorithm
- Typically the square operation is implemented differently compared with the multiply (for speed purposes)
- Then, the power trace of the exponentiation can directly yields the corresponding value
- **All programs involving conditional branching based on the key values are at risk!**

```
exp1(M, e, N)
{
    R = M
    for (i = n-2 down to 0)
    {
        R = R2 mod N
        if (ith bit of e is a 1)
            R = R · M mod N }
    return R }
```

square and multiply
algorithm



SPA examples

- Figure 11 shows a segment of the power trace of a modular exponentiation loop in which direct interpretation of the SPA features reveals an RSA decryption key.
 - This trace shows a sequence of squares and multiplications as the device performs modular exponentiation using the binary left-to-right algorithm. Multiplications consume more power than squares in this trace, and appear as by higher peaks.
 - In the binary left-to-right algorithm, one square is performed in every iteration of the exponentiation loop, while multiplications are only performed when a bit of the exponent is 1.
 - This fact allows the pattern of operations in Fig. 11 to be interpreted. Each 1 bit in the secret exponent appears as a shorter bump followed by a taller one, while a 0 bit appears as a shorter bump without a subsequent taller one. The bits of the exponent can thus be recovered as shown.

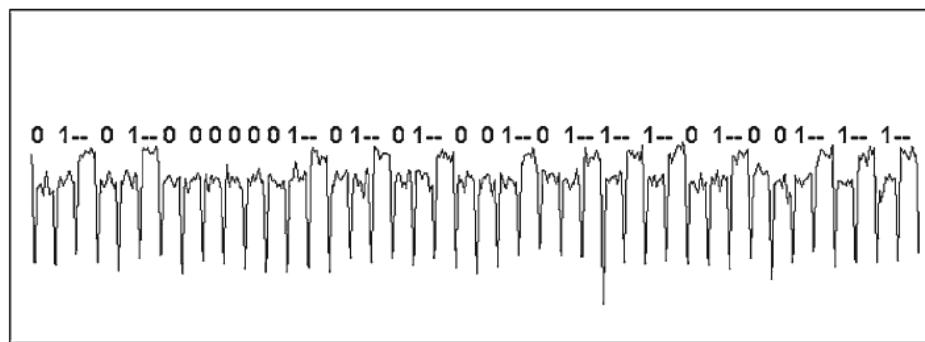
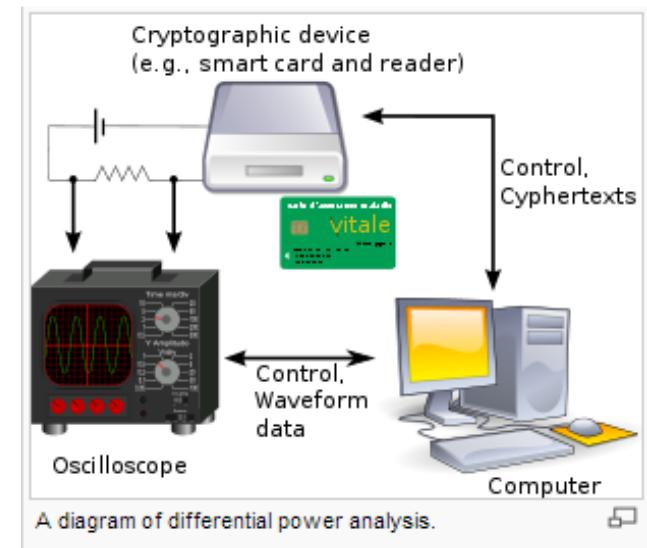


Fig. 11 SPA leaks from an RSA implementation

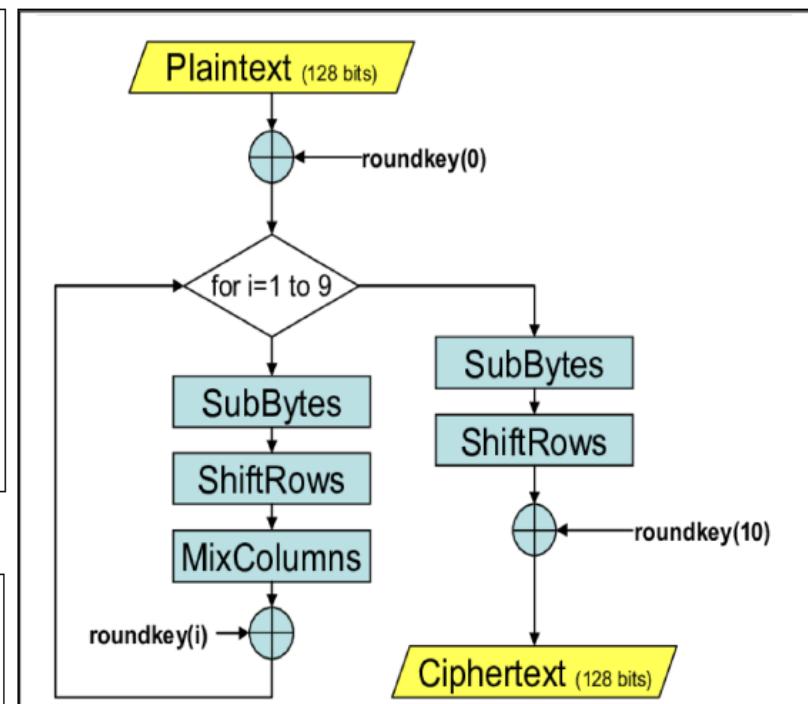
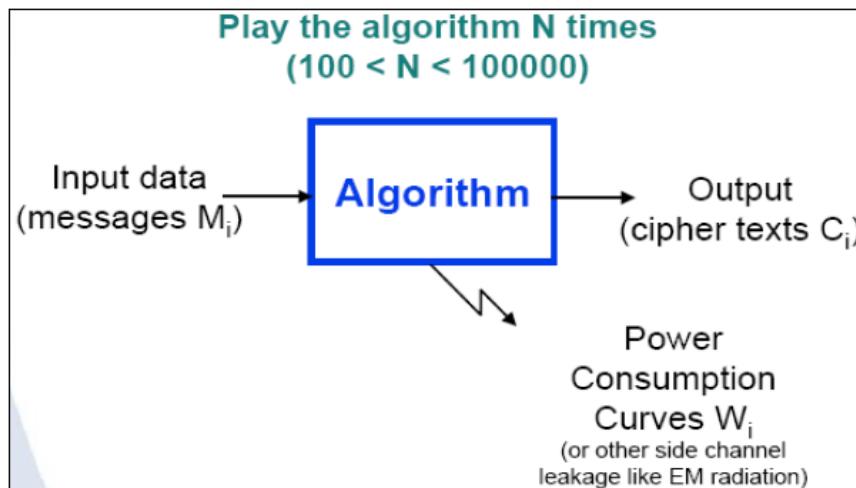
Differential Power Analysis (DPA)

- SPA targets variable instruction flow
- DPA targets data-dependence
 - Different operands present different power
- Difference between smart cards and FPGAs
 - In smart cards, **one operation running at a time**
 - ->Simple power tracing is possible
- In FPGAs, typically **parallel computations** prevent visual SPA inspection -> DPA



DPA

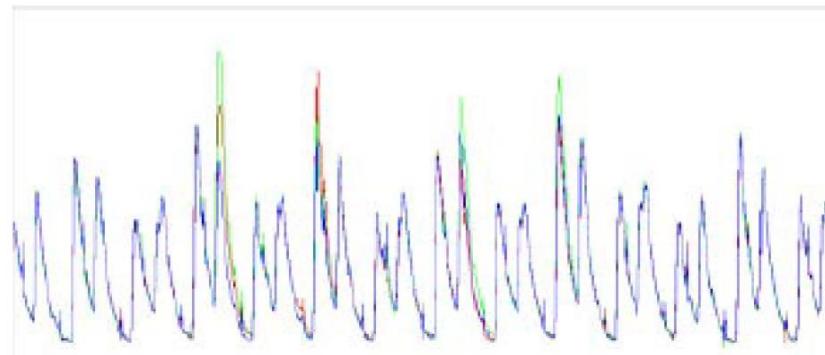
- DPA can be performed on any algorithm that has the operation $\beta = S(\alpha \oplus K)$
 - α is known and K is the segment key
 - AES example



The waveforms are captured by a scope and sent to a computer for analysis

What is available after acquisition?

- After data collection, what is available ?
 - N plain and/or cipher random texts
 - 00 B688EE57BB63E03E**
 - 01 185D04D77509F36F**
 - 02 C031A0392DC881E6 ...**
 - N corresponding power consumption waveforms



DPA

- Assume the data are processed by a known deterministic function f (transfer, permutation...)
- Knowing the data, one can re-compute off line its image through f



- Now **select** a single bit among M' bits (in M' buffer)
- One can **predict** the true story of its variations

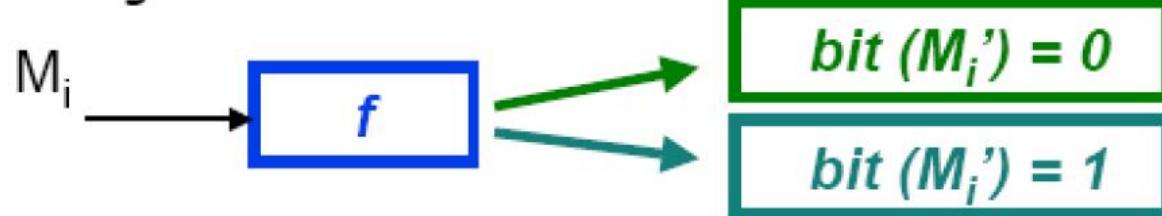
i	Message	bit
0	B688EE57BB63E03E	1	
1	185D04D77509F36F	0	
2	C031A0392DC881E6	1

The bit will classify the wave w_i

- Hypothesis 1: bit is zero
- Hypothesis 2: bit is one
- A differential trace will be calculated for each bit!

DPA

- Partition the data and related curves into two packs, according to the selection bit value...



0	B688EE57BB63E03E	1
1	185D04D77509F36F	0
2	C031A0392DC881E6	1
		...

- Sum the signed consumption curves and normalise
- \Leftrightarrow Difference of averages

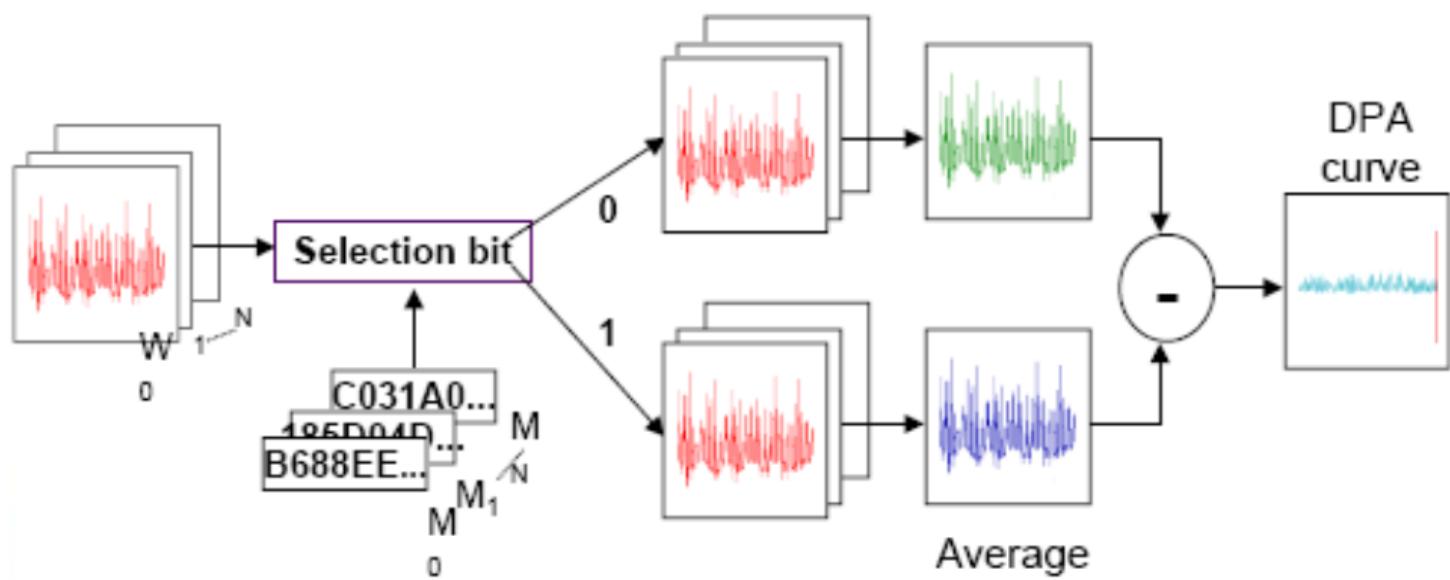
$$(N_0 + N_1 = N)$$

$$DPA = \frac{\sum W_1}{N_1} - \frac{\sum W_0}{N_0}$$

DPA Selection Function

- If the choice of which trace is assigned to each subset is uncorrelated to the measurements contained in the traces, the difference in the subsets' averages will approach zero as the number of traces increases.
- Otherwise, if the partitioning into subsets is correlated to the trace measurements, the averages will approach a nonzero value. Given enough traces, extremely tiny correlations can be isolated—no matter how much noise is present in the measurements.

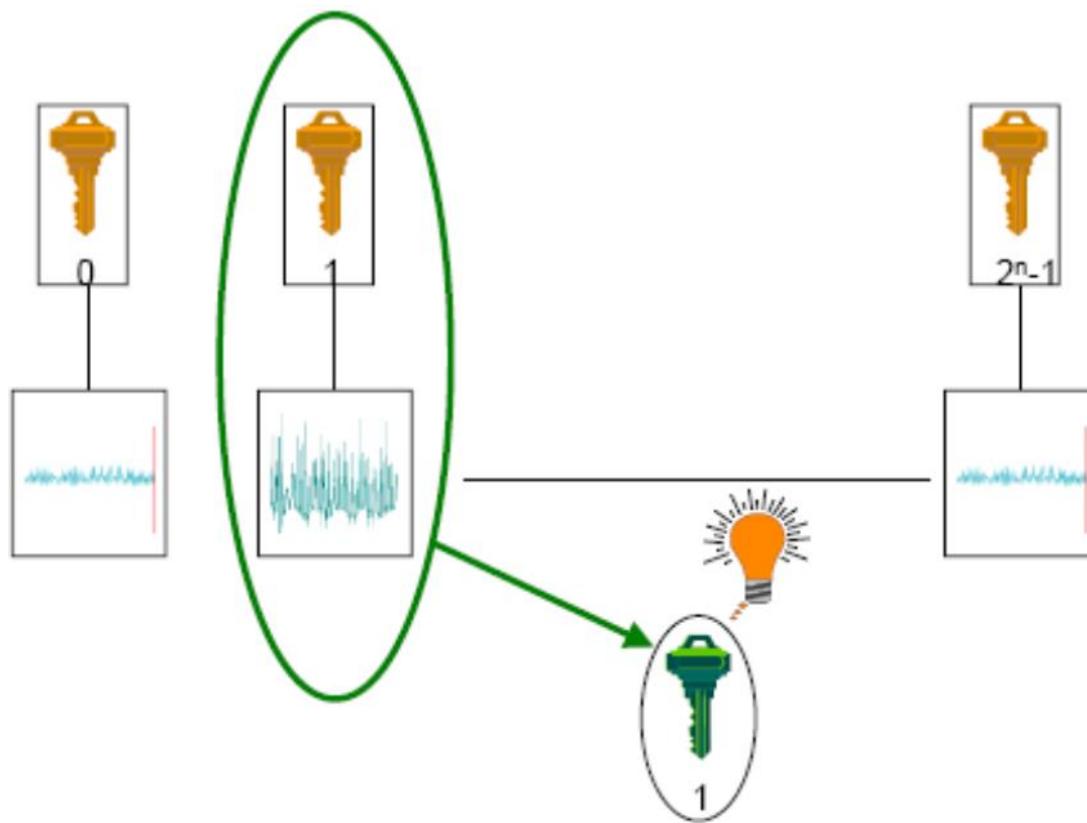
DPA



$$\Delta_n = \frac{\sum_{w_i \in S_0} w_i}{|S_0|} - \frac{\sum_{w_i \in S_1} w_i}{|S_1|}$$

DPA -- testing

- Right guess produces the highest spikes.



DPA Testing

Right guess



Exact prediction of
the selection bit

0 B688EE57BB63E03E

1 1

D = 1

1 185D04D77509F36F

0 0

D = 1

2 C031A0392DC881E6

1 1

D = 0

...

Real

Predicted

Average 1

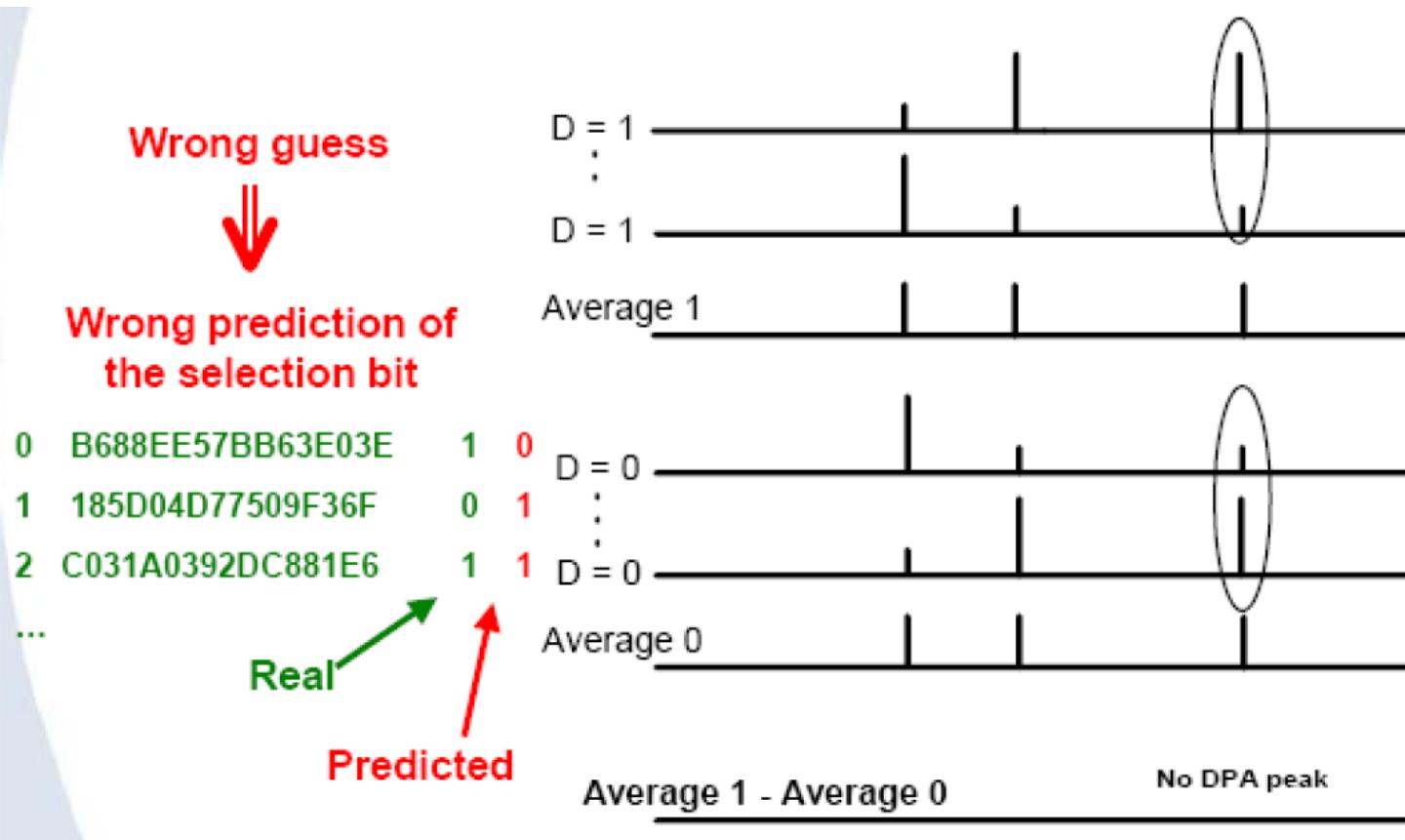
D = 0

Average 0

Average 1 - Average 0

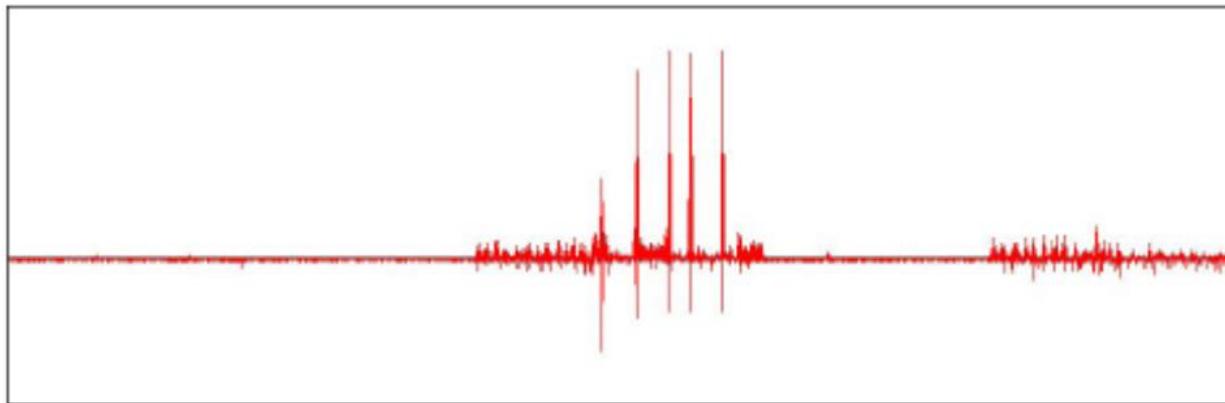
DPA peak

DPA – wrong guess



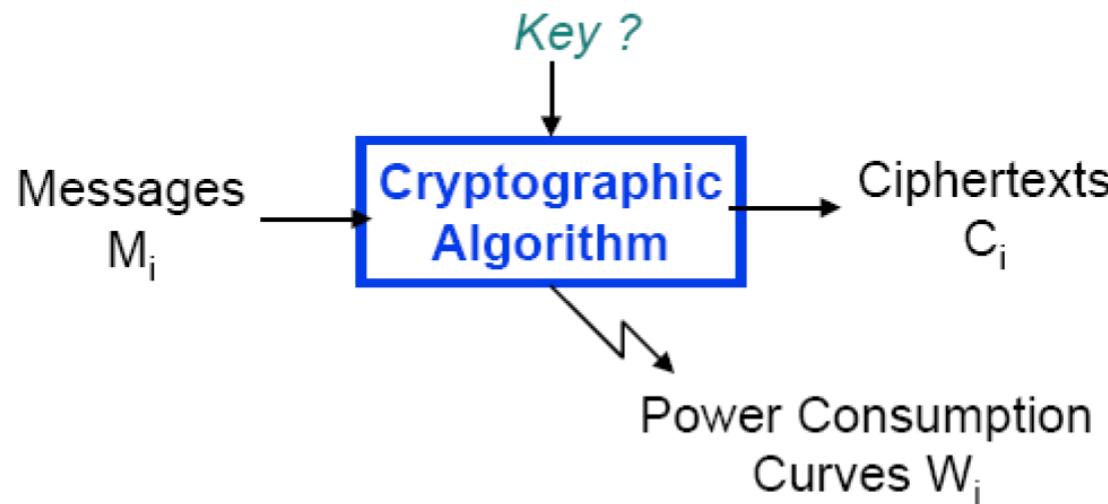
DPA

- The DPA waveform with the highest peak will validate the hypothesis



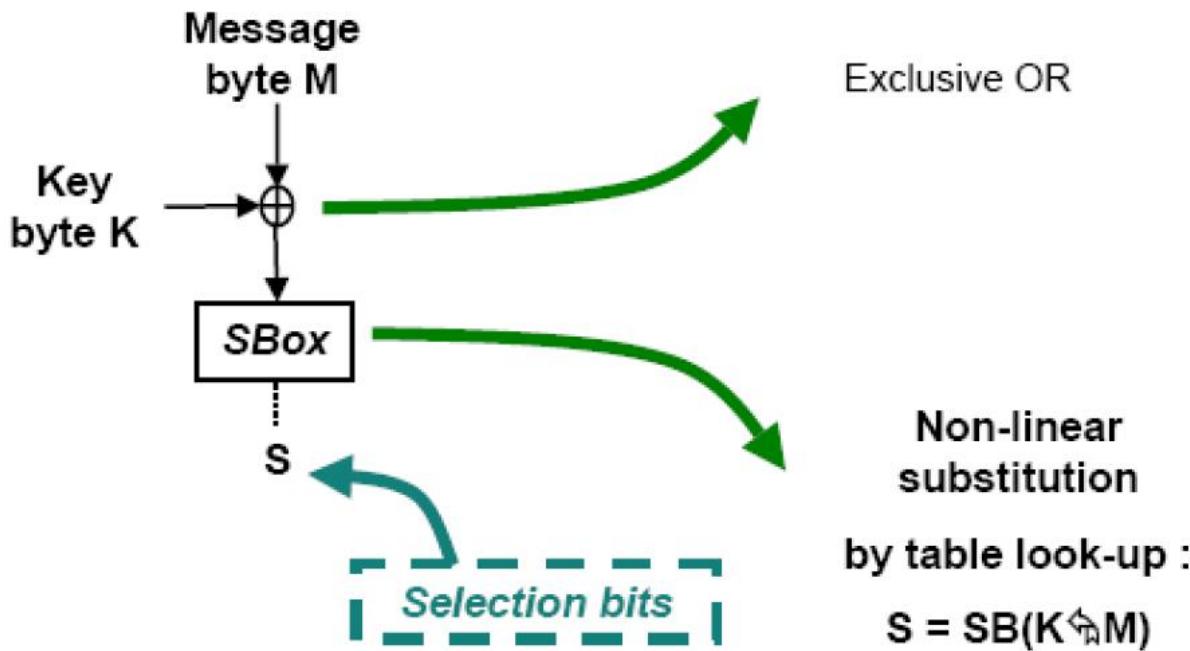
Attacking a secret key algorithm

- DPA works thanks to the perfect prediction of the selection bit
- How to break a key?



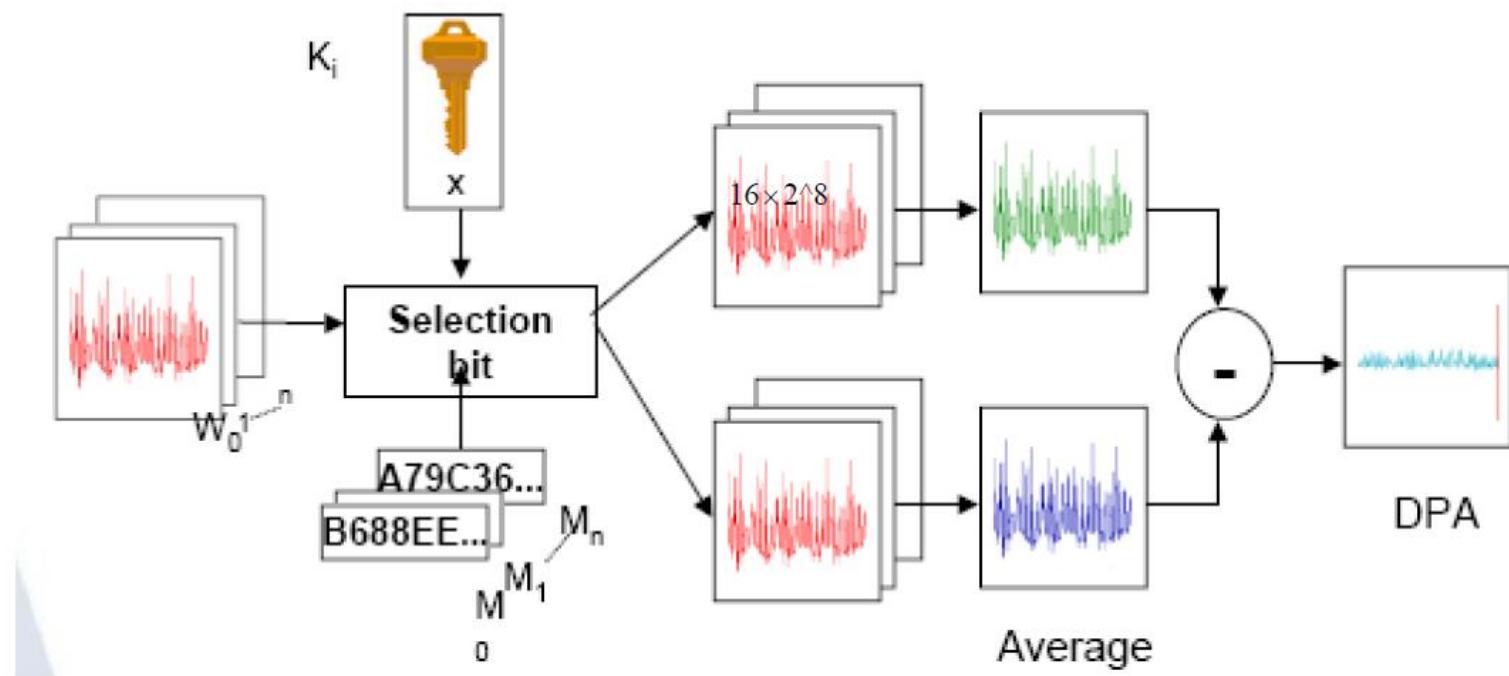
Typical DPA target

- Basic mechanism in Secret Key algorithms (AES, DES...)



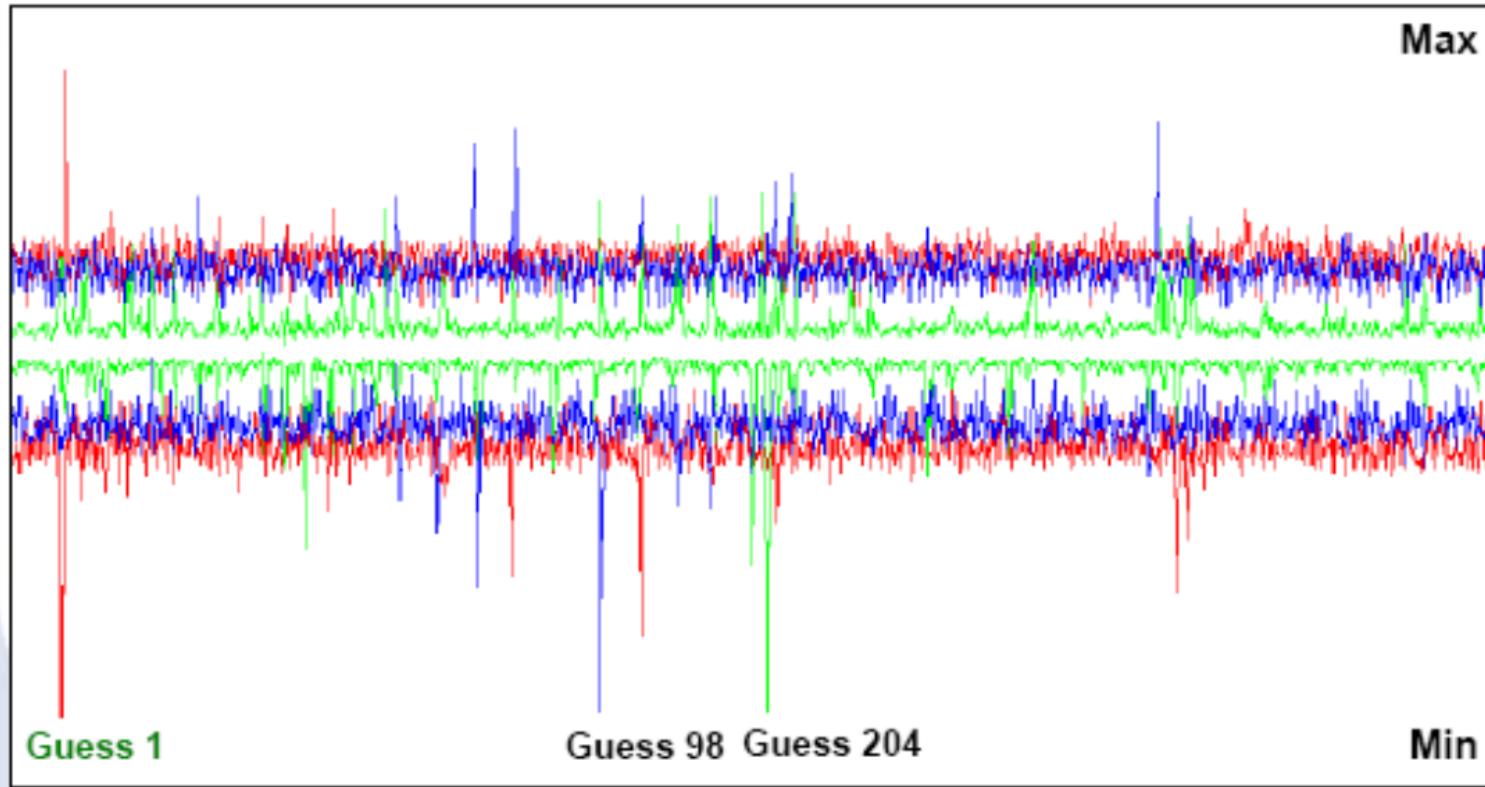
Example – DPA on AES

- AES 128 bits key = 16 bytes K_i ($i=1$ to 16)
 - Test 256 guesses per K_i with 256 DPA
 - 128 key bits disclosed with $16 \times 256 = 4096$ DPA ($<<2^{128}!$)



Example – Hypothesis testing

DPA on AES : 1st round and 1st byte (right guess = 1)



General Countermeasures

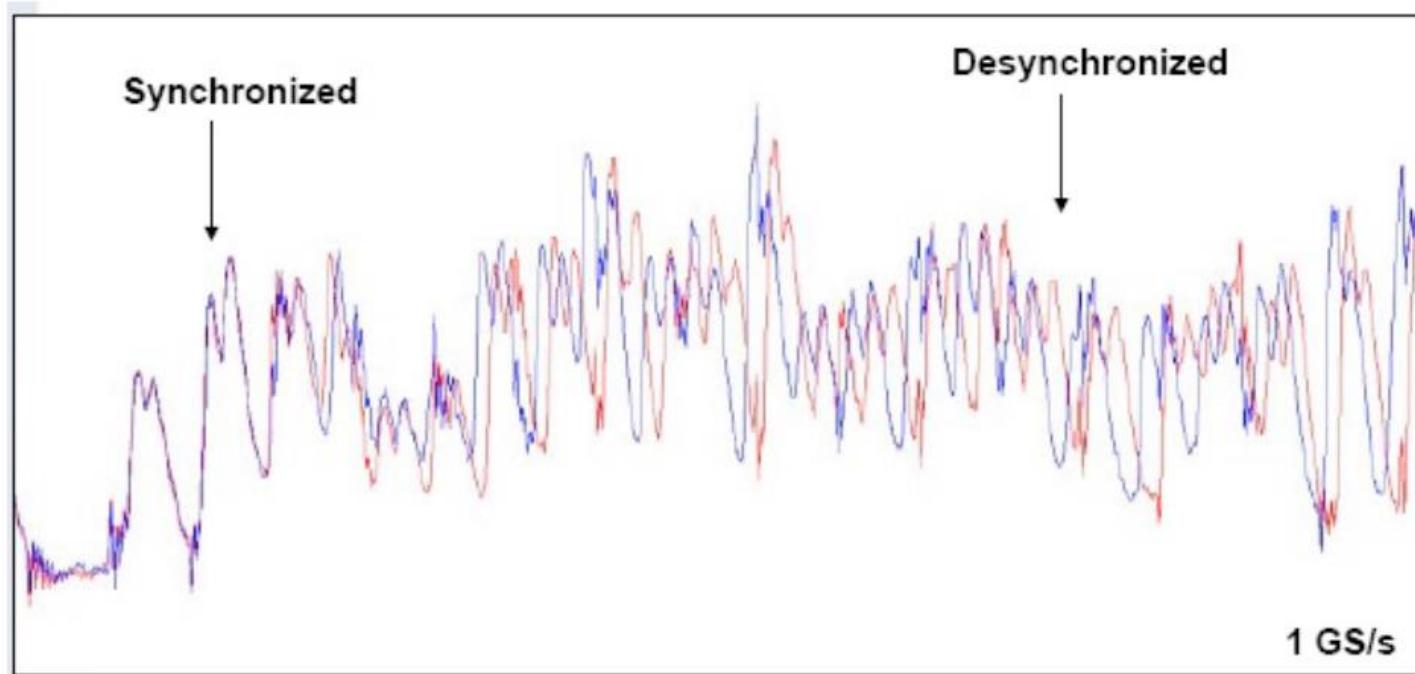
- **Hiding** -- reduce the SNR by either increasing the noise or reducing the signal
 - Noise Generators, Balanced Logic Styles, Asynchronous Logic, Low Power Design and Shielding
- **Masking/Blinding** -- remove the correlation between the input data and the side-channel emissions from intermediate nodes in the functional block
- **Design Partitioning** -- separate regions of the chip that operate on plaintext from regions that operate on ciphertext
- **Physical Security and Anti-Tamper** -- denial of proximity, access, and possession

Anti-DPA countermeasures

- Decorrelate power curves from data
 - By hardware: current scramblers (additive noise)
 - By software: data whitening
- Desynchronize N traces (curves misalignment)
 - Software random delays
 - Software random orders (ex: Sboxes in random order)
 - Hardware wait states (dummy cycles randomly added by CPU)
 - Hardware unstable internal clock (phase shift)
- DPA is powerful, generic (to many algorithms) and robust to model errors
 - But there are counter-measures

Anti DPA

- Internal clock phase shift



ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 18: Hardware Trojans

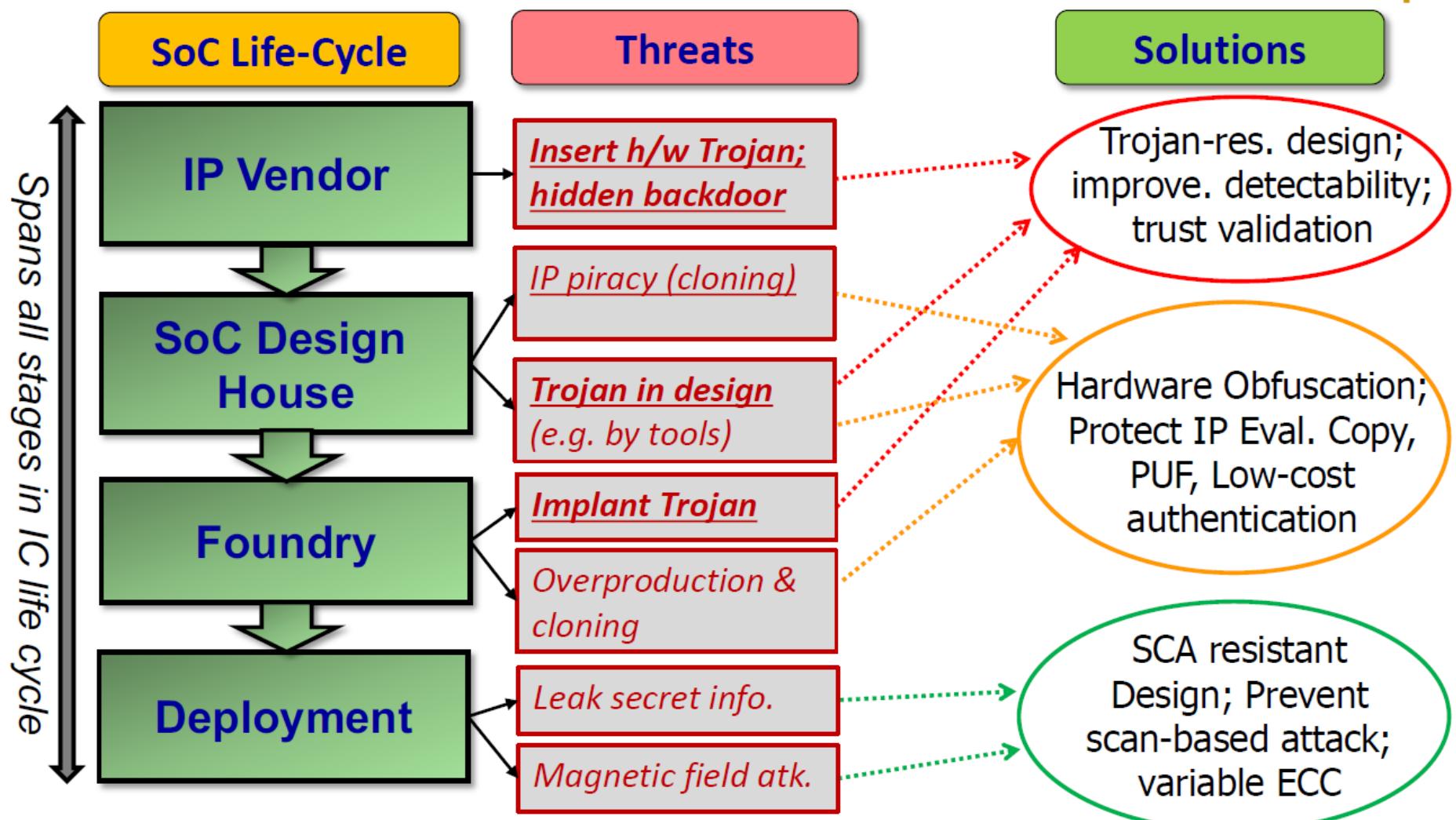
04/10/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Slides are adapted from Mark Tehranipoor, U. of Florida

Threats



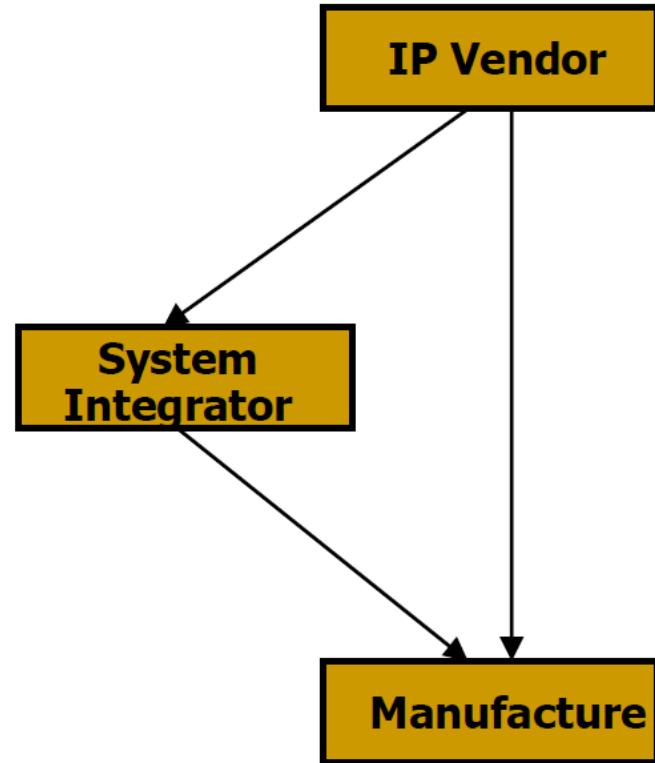
What is Hardware Trojan?

- **Hardware Trojan:**
 - A malicious addition or modification to the existing circuit elements.
- **What hardware Trojans can do?**
 - Change the functionality
 - Reduce the reliability
 - Leak valuable information
- **Applications that are likely to be targets for attackers**
 - Military applications
 - Aerospace applications
 - Civilian security-critical applications
 - Financial applications
 - Transportation security
 - IoT devices
 - Commercial devices
 - More

IC/IP Trust Problem

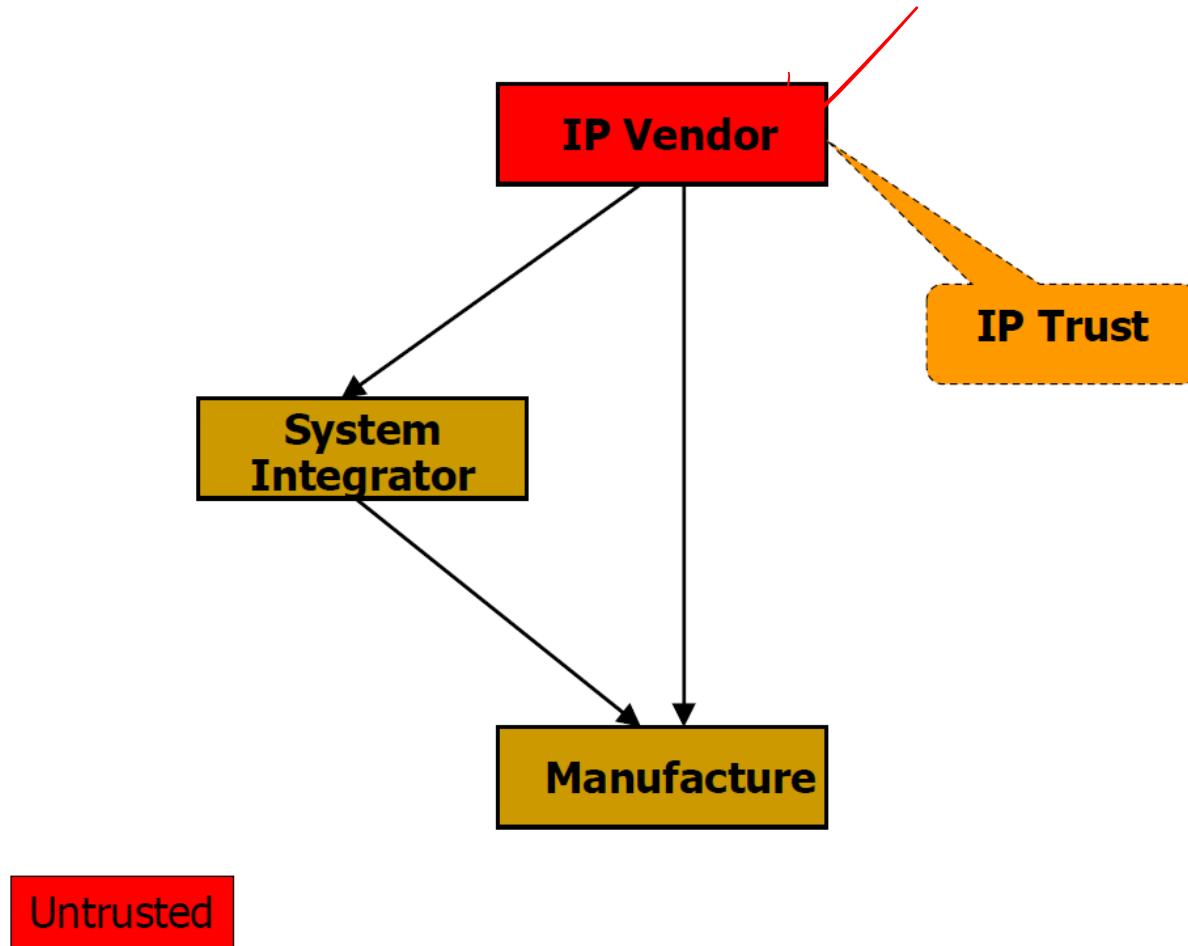
- Chip design and fabrication has become increasingly vulnerable to malicious activities and alterations with globalization.
- **IP Vendor and System Integrator:**
 - IP vendor may place a Trojan in the IP
 - *IP Trust problem*
- **Designer and Foundry:**
 - Foundry may place a Trojan in the layout sign.
 - *IC Trust problem*

Hardware Trojan Threat



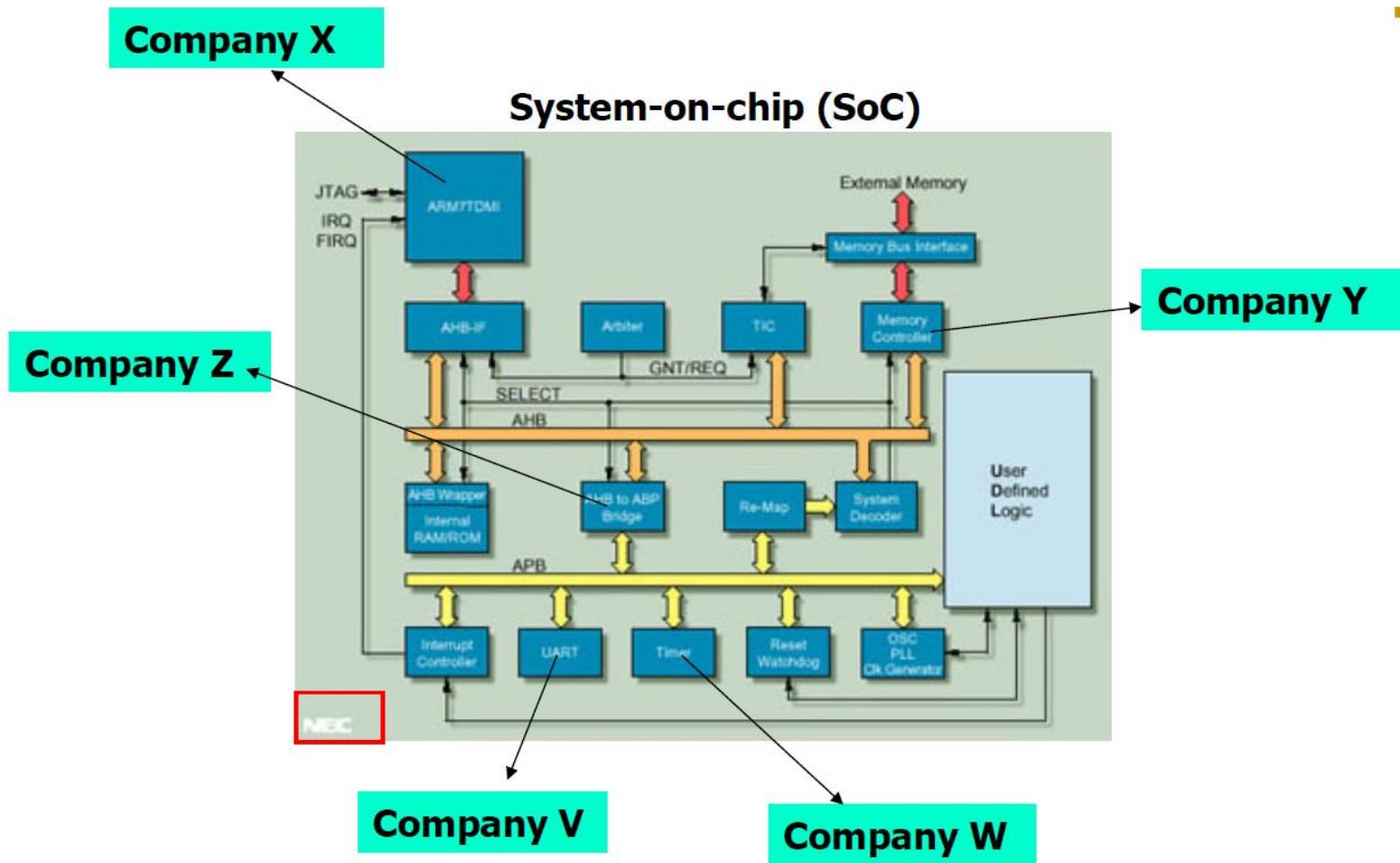
Any of these steps can be untrusted

Hardware Trojan Threat

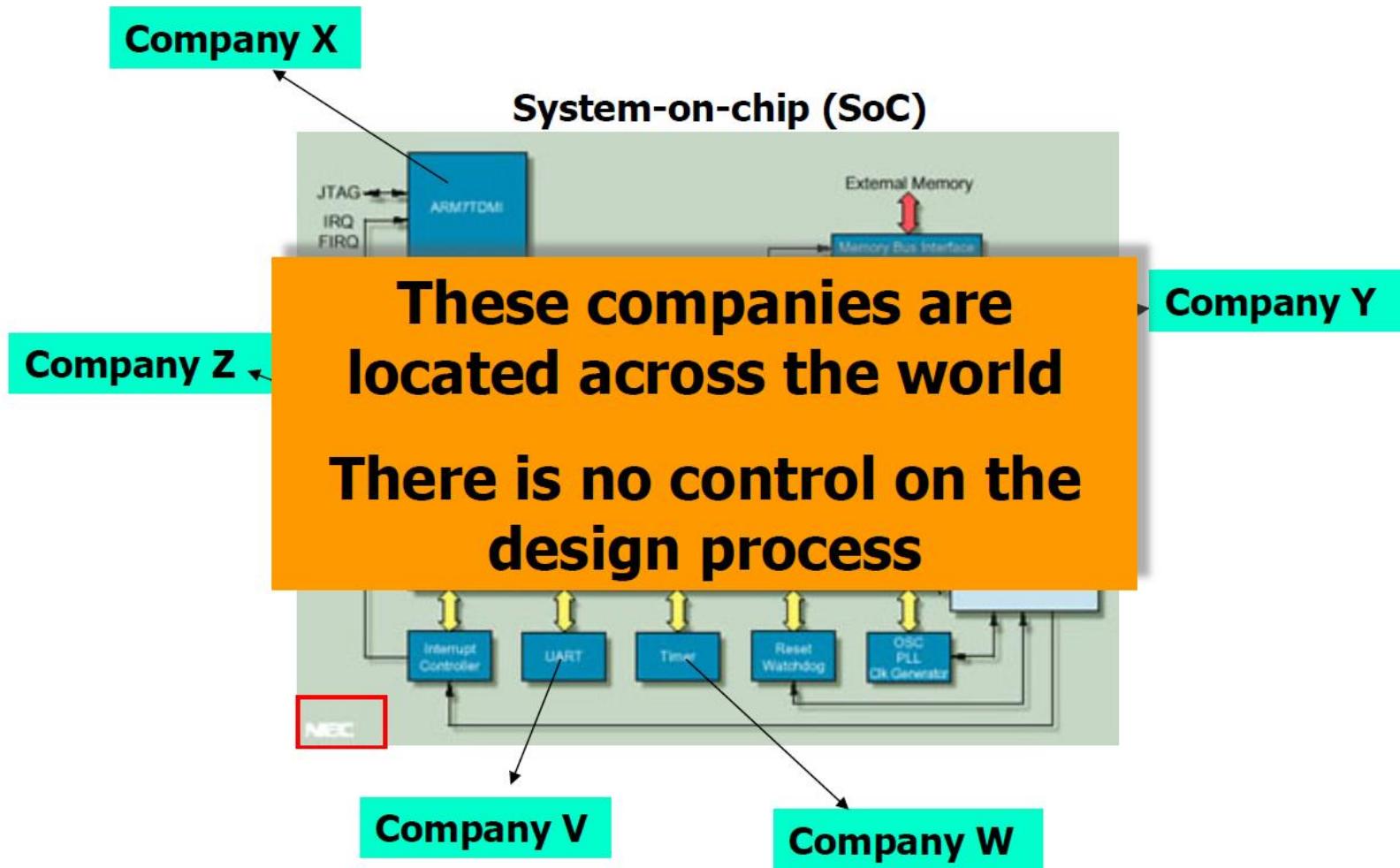


Untrusted

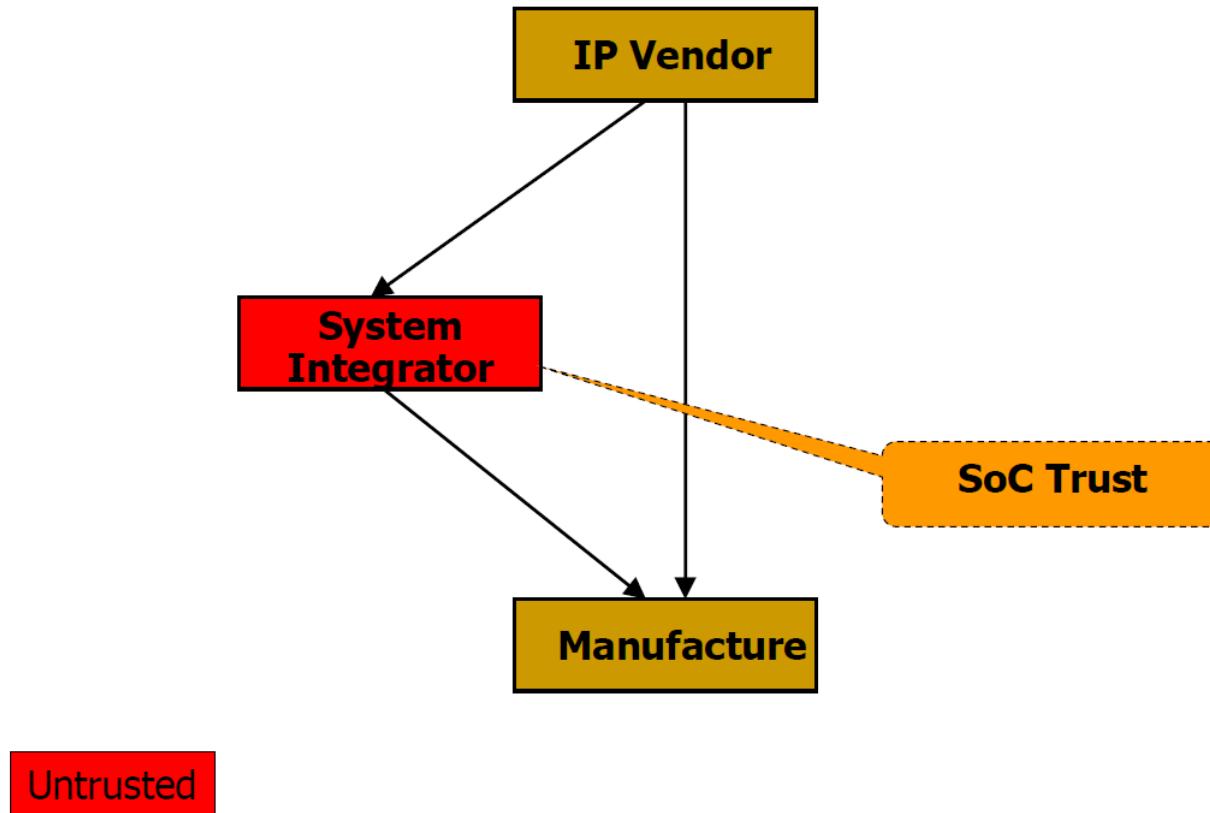
Issues with Third IP Design



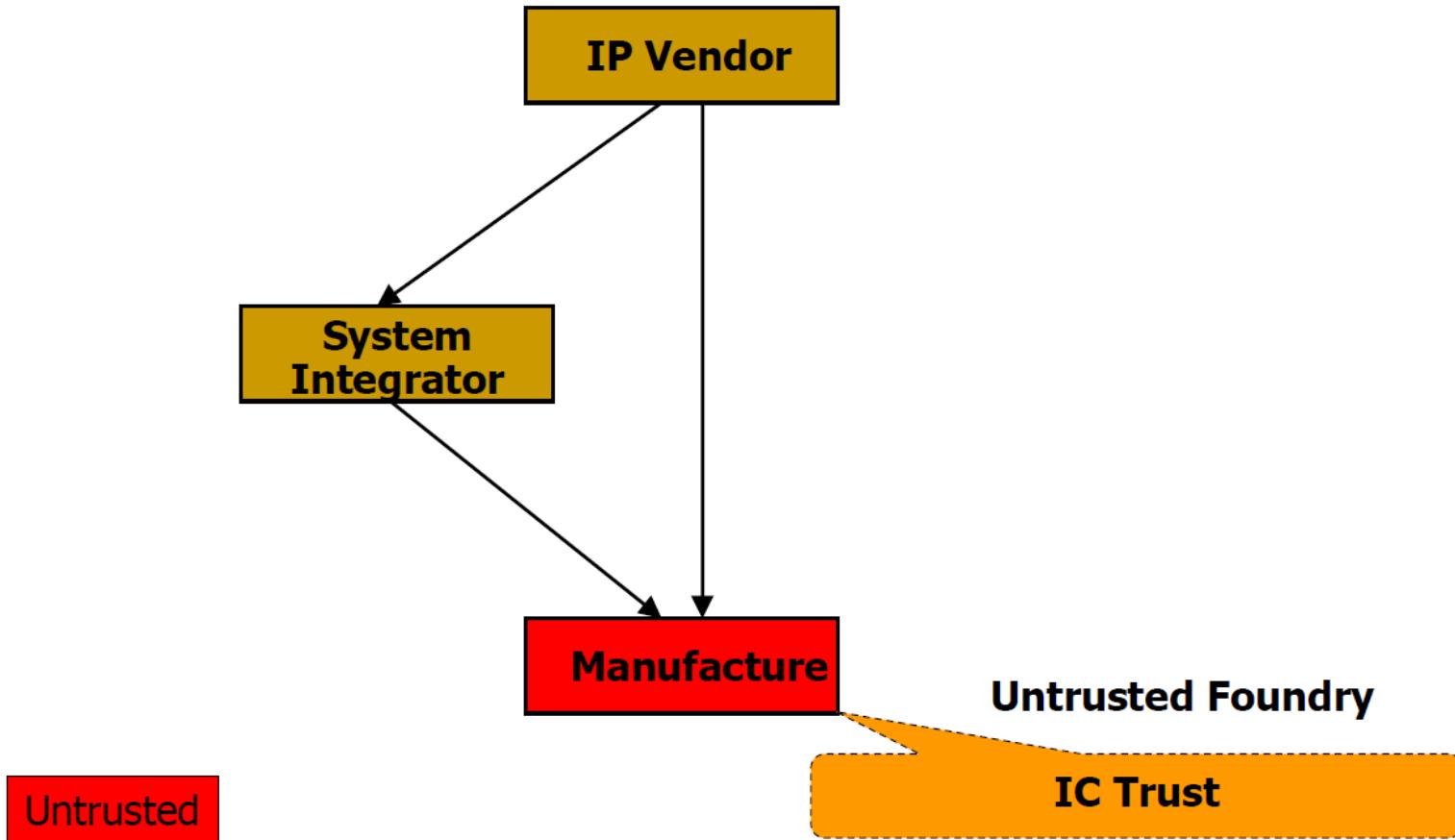
Issues with Third IP Design



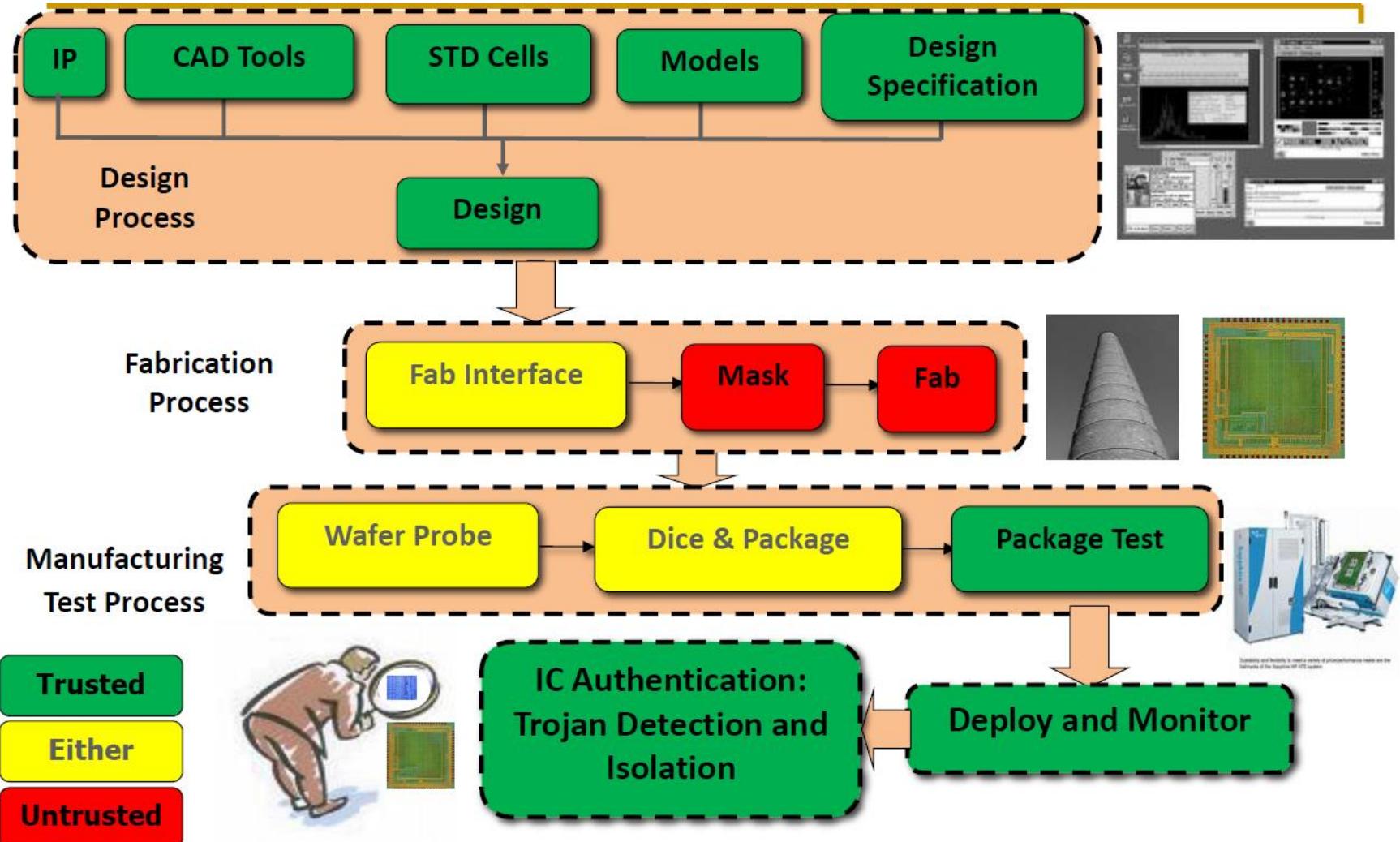
Hardware Trojan Threat



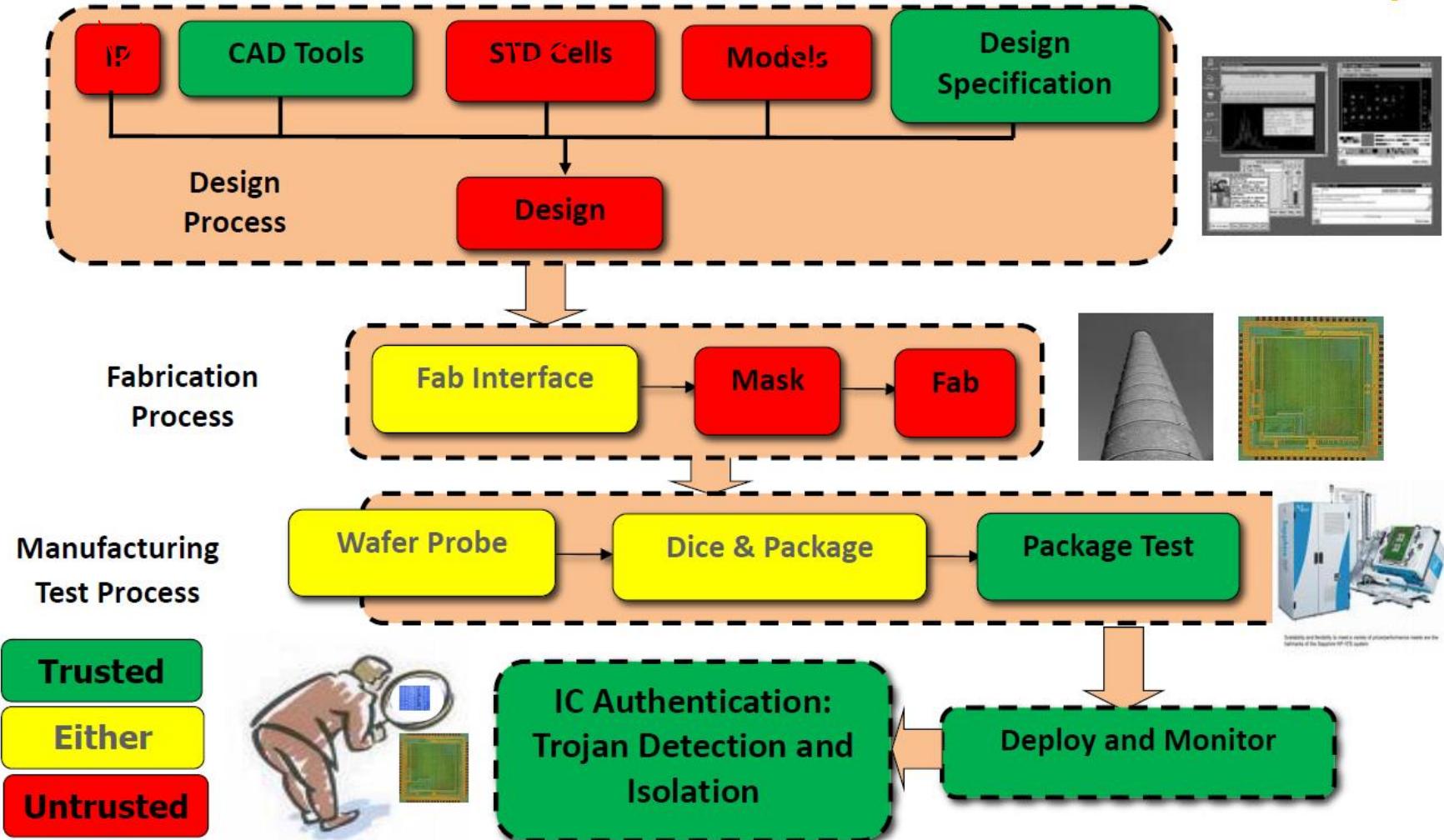
Hardware Trojan Threat



ASIC Design Process- Untrusted Foundry

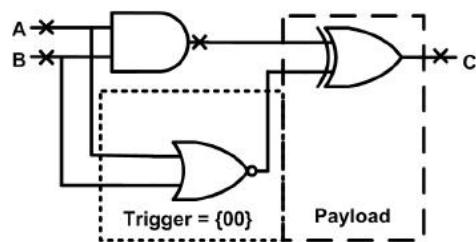


Untrusted Designer and Foundry

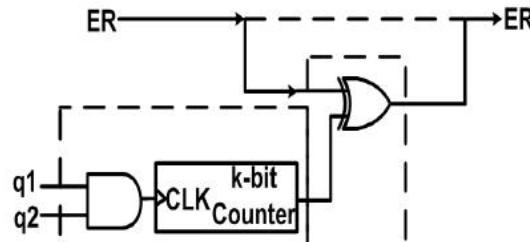


HW Trojan Examples / Models

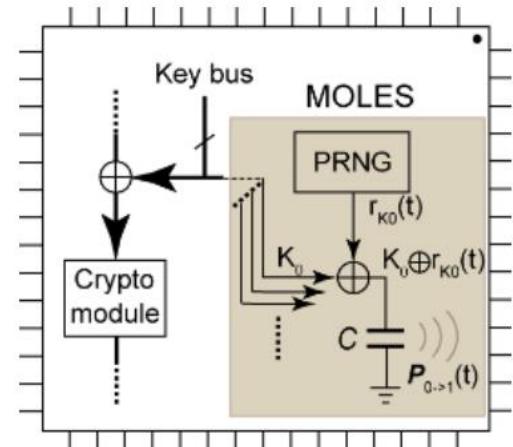
Comb. Trojan Example



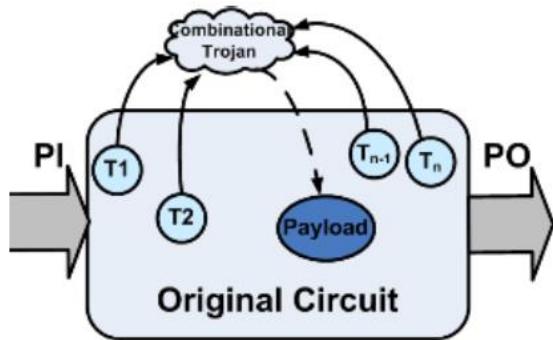
Seq. Trojan Example



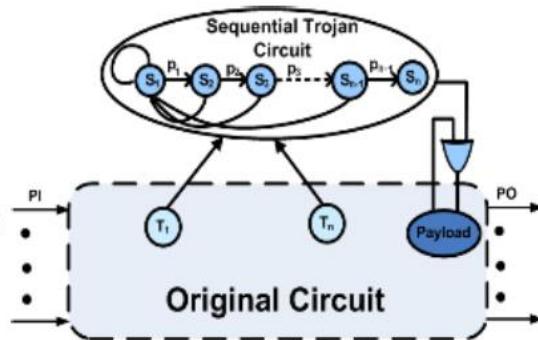
MOLES: Info Leakage Trojan*



Comb. Trojan model



Seq. Trojan Model



*Lin et al, ICCAD 2009

Fishy Chips: Spies Want to Hack-Proof Circuits

By Adam Osborne
08:24am
12/10/12
Palo Alto



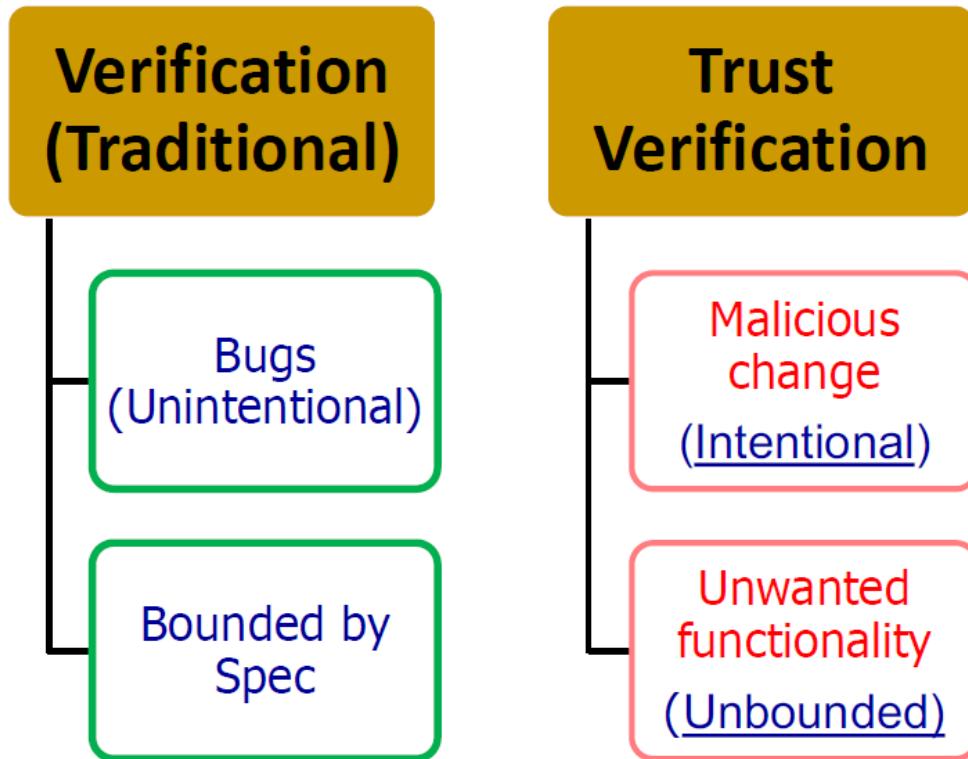
In 2009, the U.S. military had a problem. It had bought over [several million](#) chips for its defense systems every year from a Chinese company to protect them from hacking. The chips turned out to be counterfeits from China, but it could have been even worse. Instead of copying Chinese fakes, being put into Navy weapon systems, the chips could have been hacked, able to shut off a missile in the event of war or be armed just waiting to detonate.

HW Trojan evidence!

Hardware Trojans

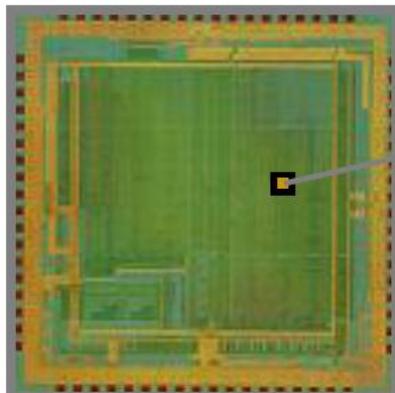
**Why is detection of hardware Trojans
very difficult?**

Bug vs. Malicious Change

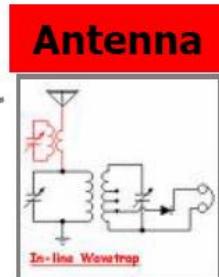


Trojan Attacks → BIGGER verification challenge!

Silicon Back Door



Untrusted Hardware



- Adversary can send and receive secret information
- Adversary can disable the chip, blowup the chip, send wrong processing data, impact circuit information etc.

- Adversary can place an Antenna on the fabricated chip
- Such Trojan cannot be detected since it does not change the functionality of the circuit.



Silicon Time Bomb



Untrusted Hardware



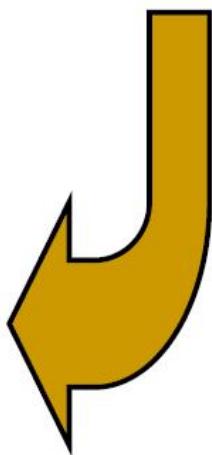
Counter

Finite state machine (FSM)

Comparator to monitor key data



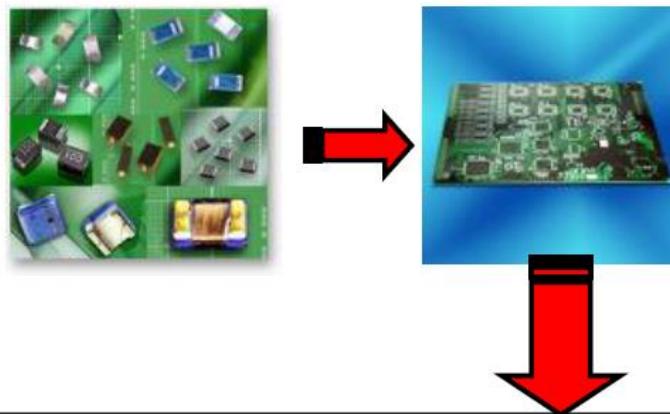
Wires/transistors that violate design rules



- Such Trojan cannot be detected since it does not change the functionality of the circuit.
- In some cases, adversary has little control on the exact time of Trojan action
- Cause reliability issue

Applications and Threats

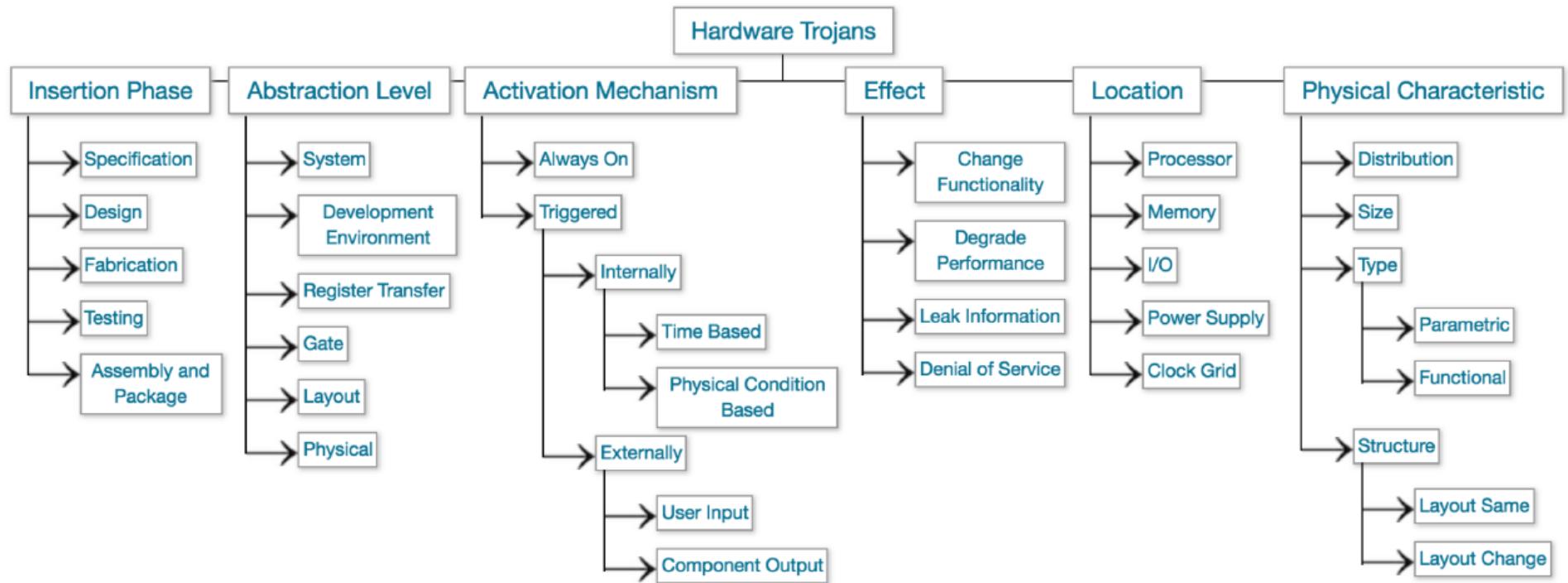
Thousands of chips are being fabricated in untrusted foundries



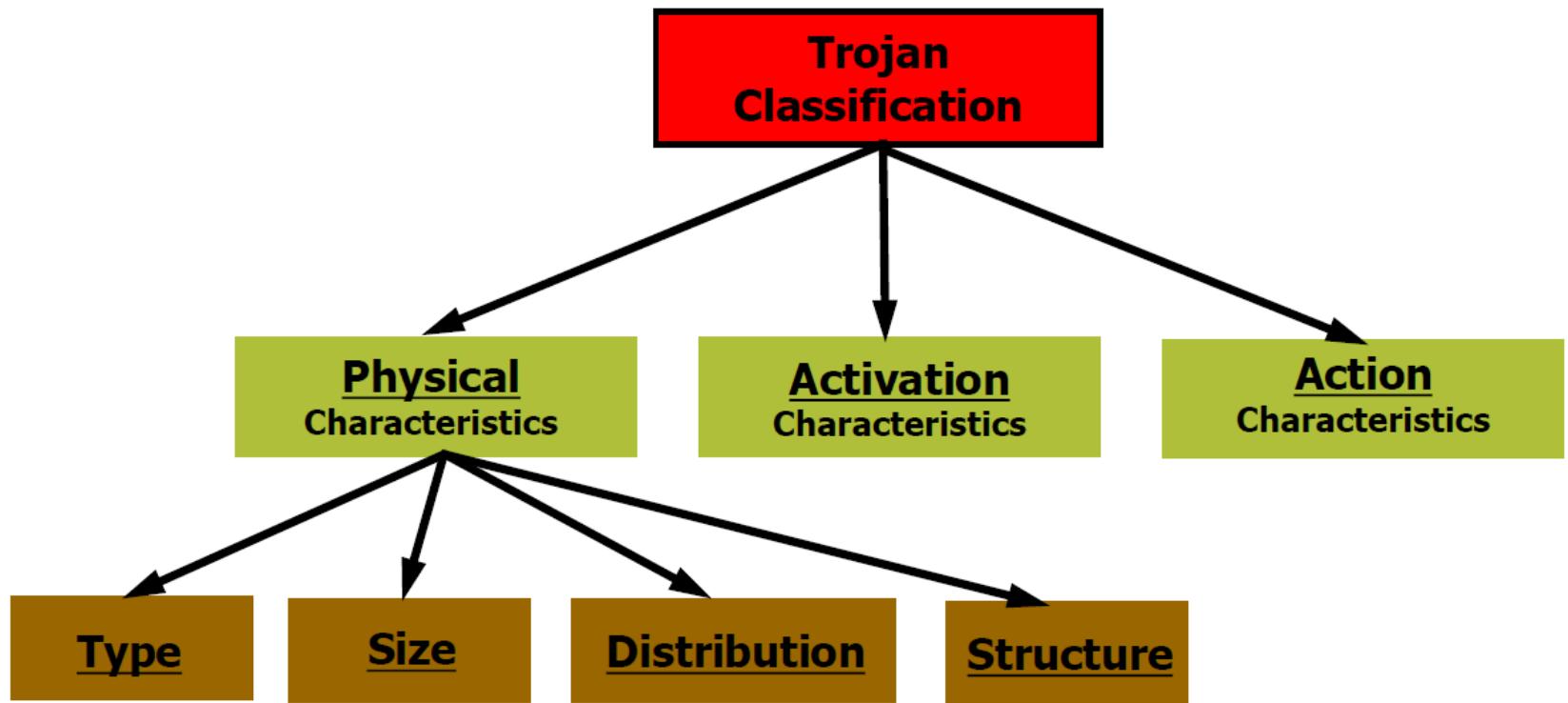
Comprehensive Attack Model

Model	Description	3PIP Vendor	SoC Developer	Foundry
A	Untrusted 3PIP vendor	Untrusted	Trusted	Trusted
B	Untrusted foundry	Trusted	Trusted	Untrusted
C	Untrusted EDA tool or rogue employee	Trusted	Untrusted	Trusted
D	Commercial-off-the-shelf component	Untrusted	Untrusted	Untrusted
E	Untrusted design house	Untrusted	Untrusted	Trusted
F	Fabless SoC design house	Untrusted	Trusted	Untrusted
G	Untrusted SoC developer with trusted IPs	Trusted	Untrusted	Untrusted

Trojan Taxonomy



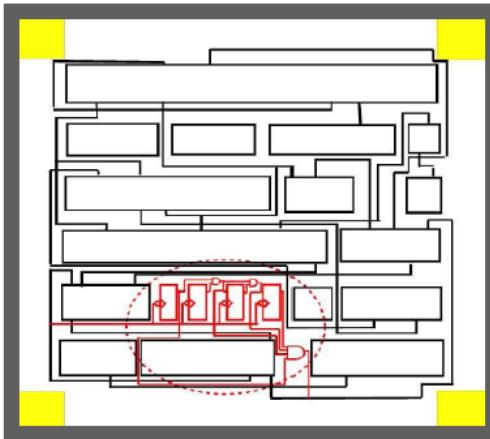
Trojan Taxonomy



Examples for Layout Level Trojans

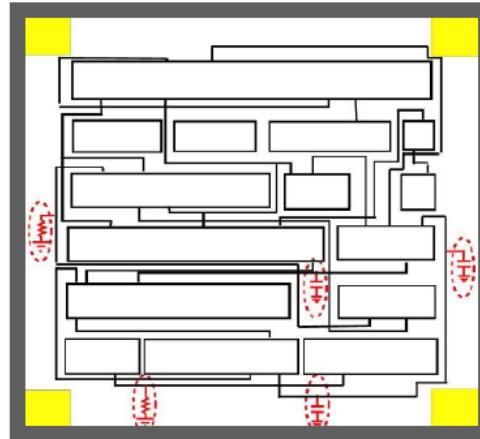
Example: Type

Functional



- **Functional** 
 - Addition or deletion of components
 - Sequential circuits
 - Combinational circuits
 - Modification to function or no change

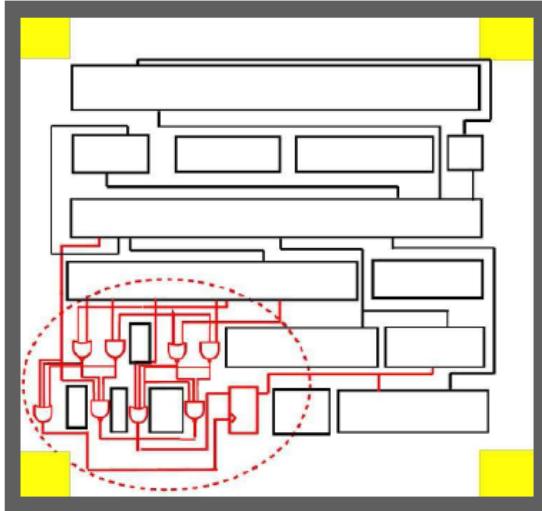
Parametric



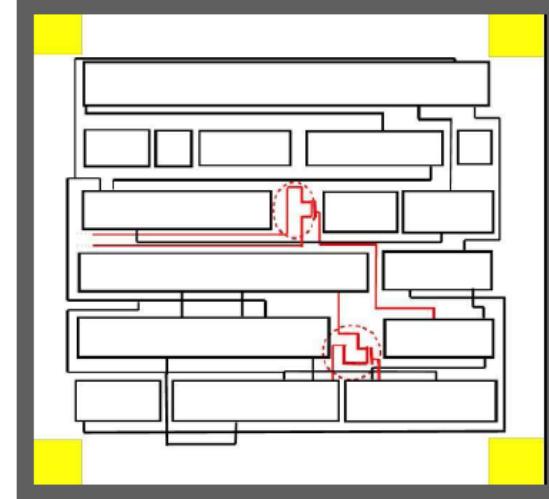
- **Parametric** 
 - Modifications of existing components
 - Wire: e.g. thinning of wires
 - Logic: Weakening of a transistor, modification to physical geometry of a gate
 - Modification to power distribution network
 - Sabotage reliability or increase the likelihood of a functional or performance failure

Example: Size

Big

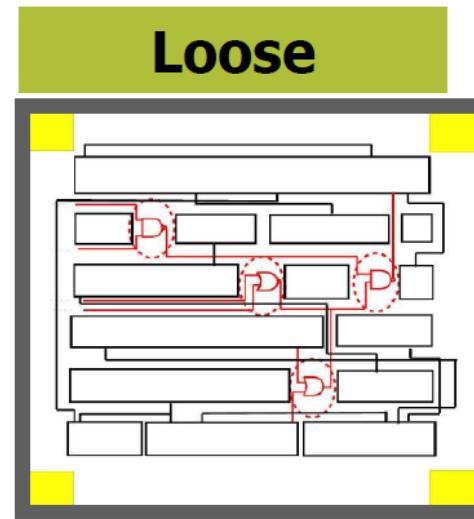
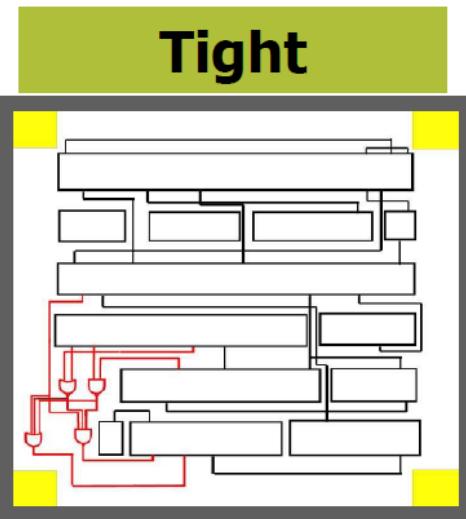


Small



- **Size:**
 - Number of components added to the circuit
 - Small transistors
 - Small gates
 - Large gates
- In case of layout, depends on availability of:
 - Dead spaces
 - Filler cells
 - Decap cells
 - Change in the structure

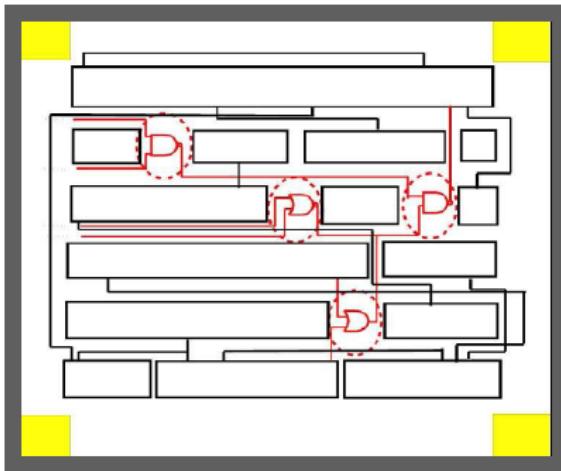
Example: Distribution



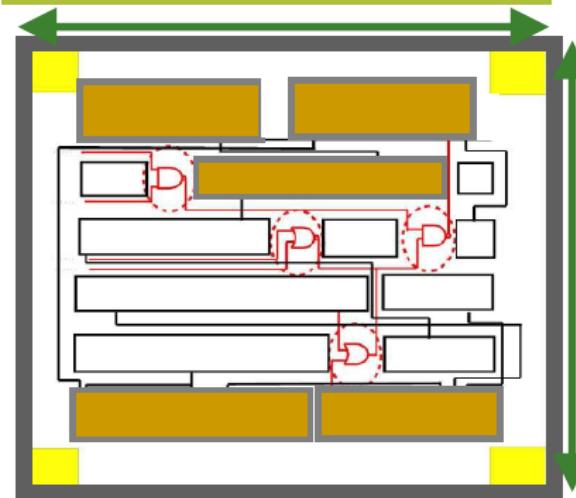
- **Tight Distribution**
 - Trojan components are topologically close in the layout
- **Loose Distribution**
 - Trojan components are dispersed across the layout of a chip
- ▶ **Distribution of Trojans depends on the availability of dead spaces on the layout**

Example: Structure

No-change



Modified Layout

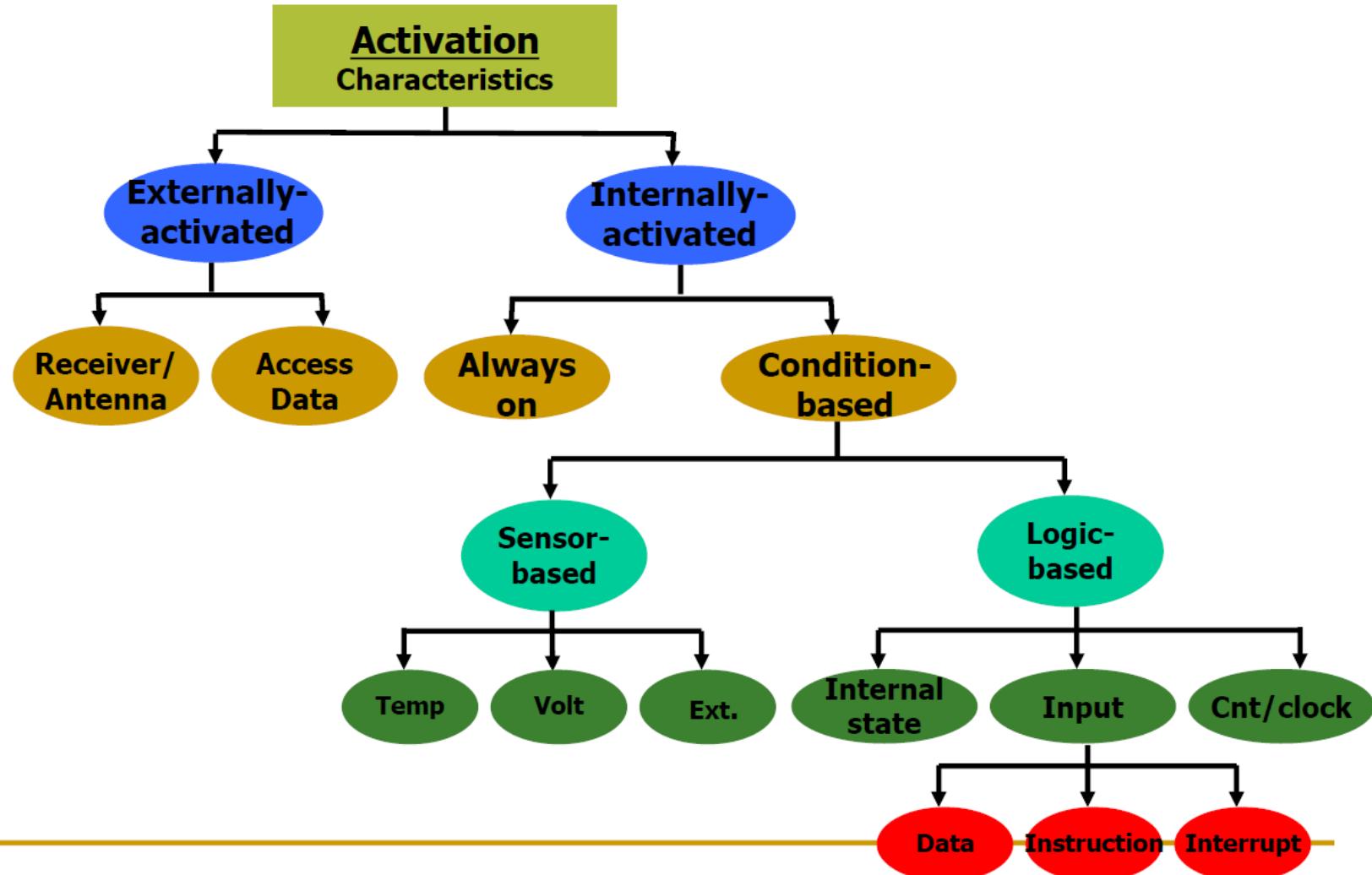


Change in circuit
Form Factor

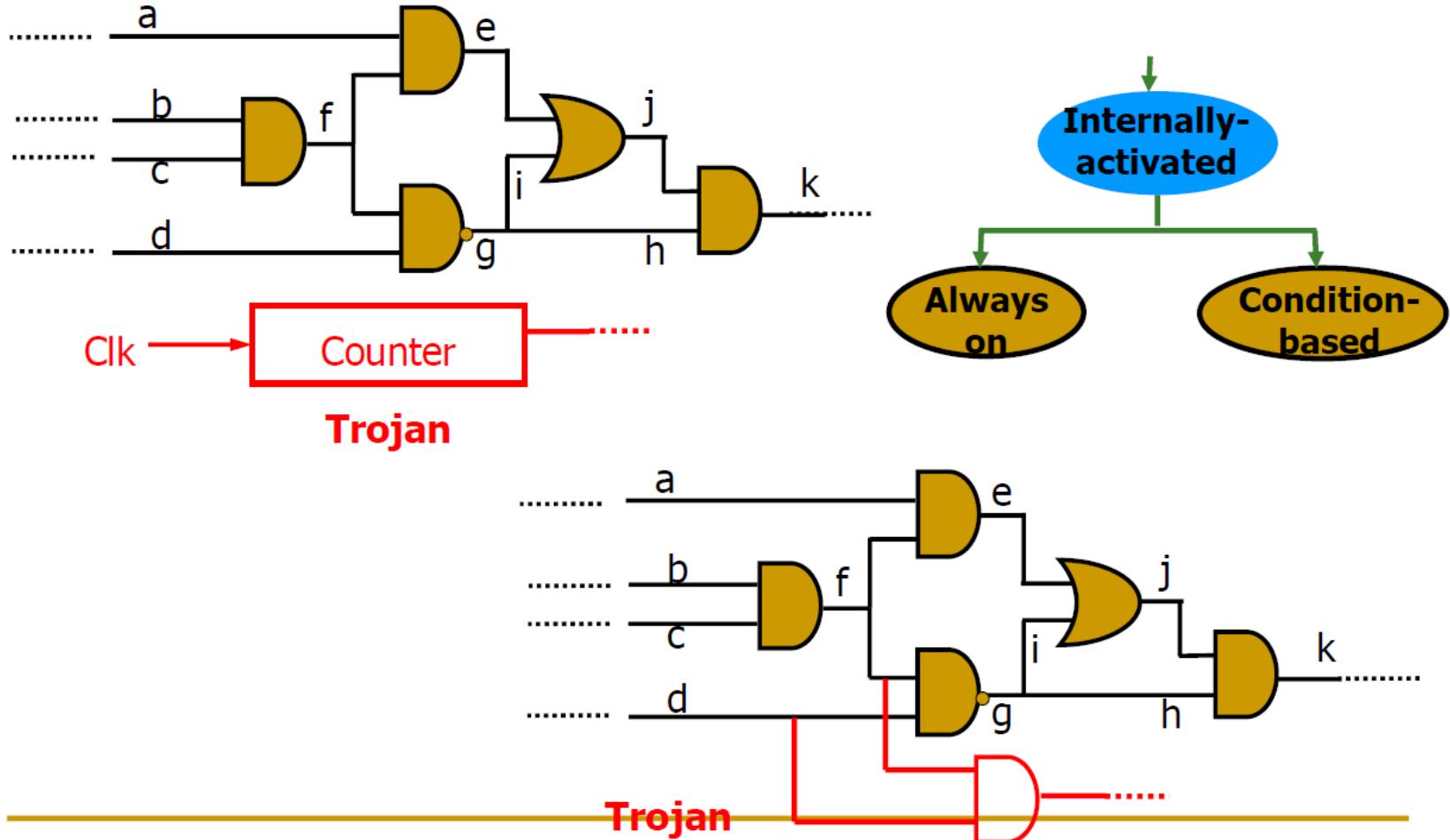
- The adversary may be forced to regenerate the layout to be able to insert the Trojan, then the chip dimensions change
 - It could result in different placement for some or all the design components

- ▶ A change in physical layout can change the delay and power characteristics of chip
 - ▶ It is easier to detect the Trojan

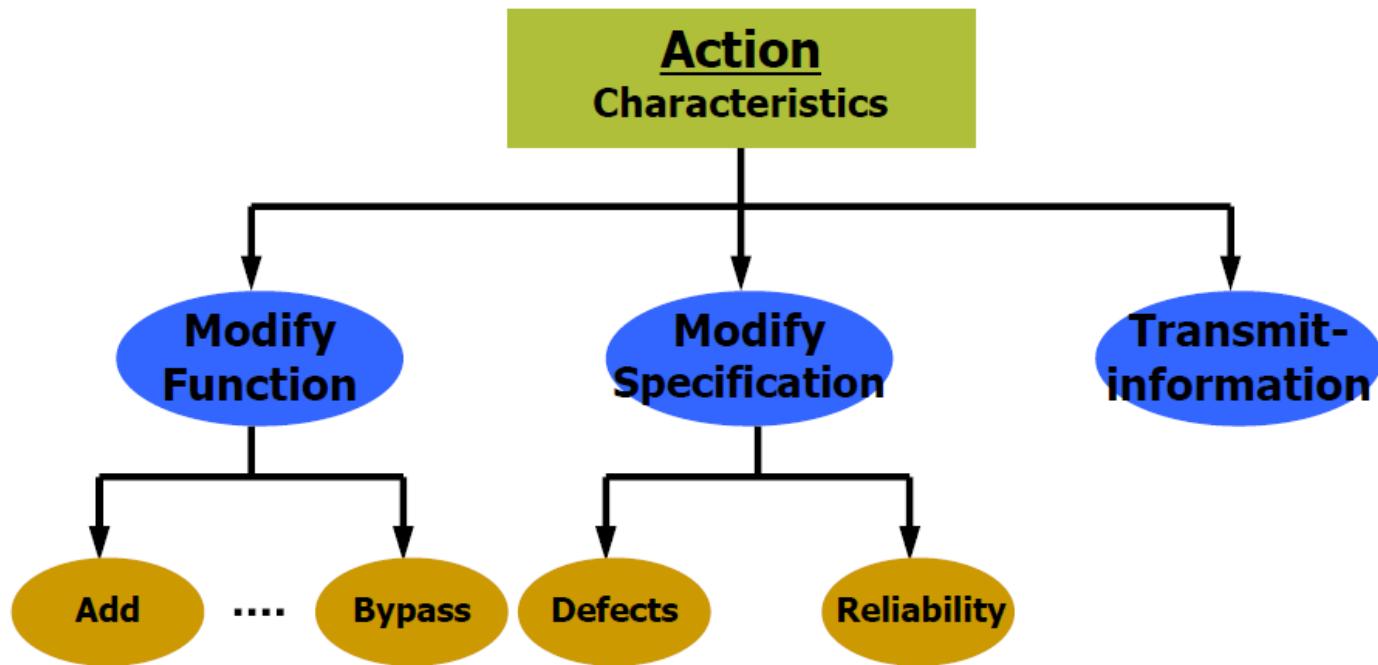
Trojan Taxonomy: Activation



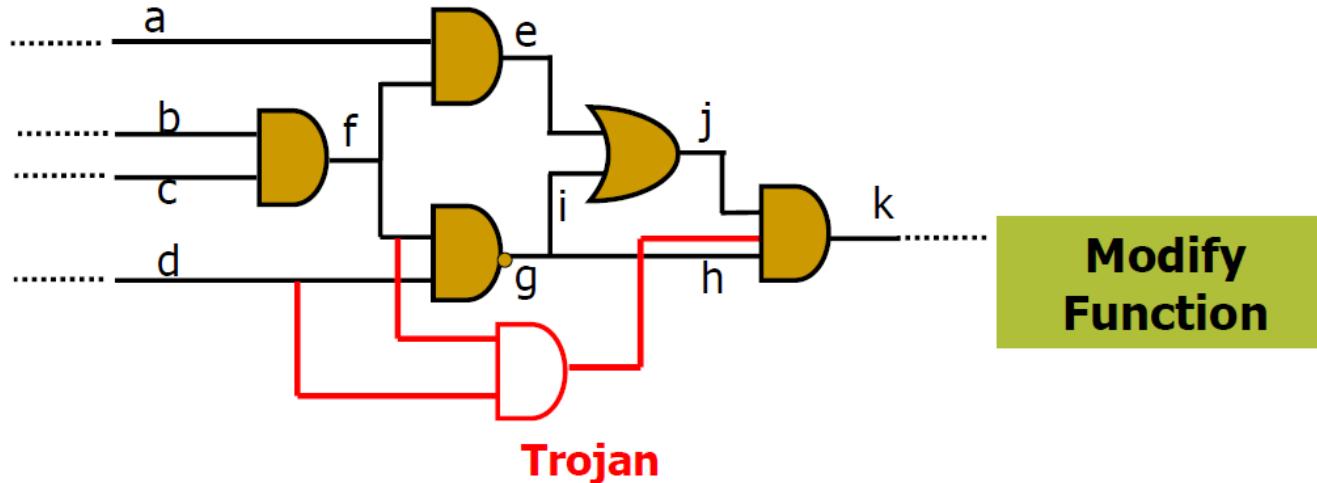
Activation: Internally Activated



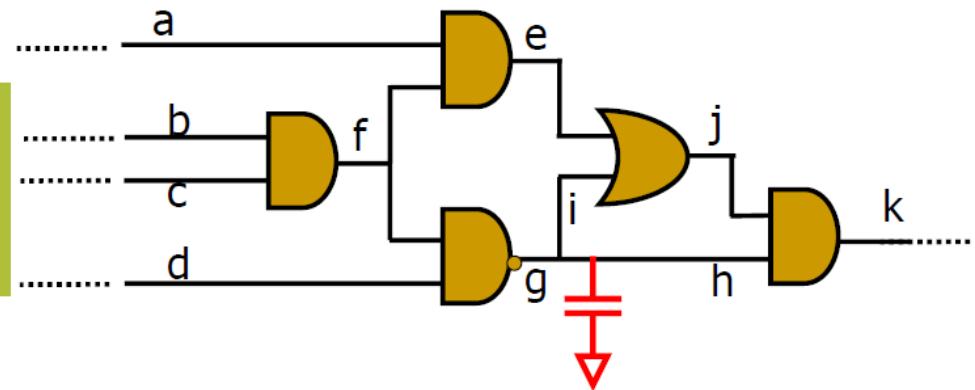
Trojan Taxonomy: Action



Example: Action



Modify Specification:
Noise, Delay and
Temperature



IP Trust and IP Security

■ IP Trust

- Detect ***malicious*** circuits inserted by IP designers
 - **Goal to Verify Trust:** Protect IP buyers, e.g., SoC integrators

■ IP Security

- Information leakage, side-channel leakage, backdoors, functional bugs and flaws, illegal IP use/overuse, etc.
 - **Goal to Verify Security:** Protect application

IP Trust

- IPs from untrusted vendors need to be verified for trust before use in a system design
- **Problem statement:** How can one establish that the IP does exactly as the specification, nothing less, nothing more?
- **IP Cores:**
 - Soft IP, firm IP and hard IP
- **Challenges:**
 - No known golden model for the IP
 - Spec could be assumed as golden
 - Soft IP is just a code so that we cannot read its implementation

Approaches for Pre-synthesis

- **Formal verification**

- Property checking
- Model checking
- Equivalence checking

- **Coverage analysis**

- Code coverage
- Functional coverage

Formal Verification

▪ **Formal verification**

- Ensuring IP core is exactly same as its specification
- Three types of verification methods
 - **Property checking:** Every *requirement* is defined as assertion in testbench and is checked
 - **Equivalence checking:** Check the equivalence of RTL code, gate-level netlist and GDSII file
 - **Model checking**
 - System is described in a formal model (C, HDL)
 - The desired behavior is expressed as a set of properties
 - The specification is checked against the model

Coverage Analysis

▶ Code coverage

▶ Line coverage

Show which lines of the RTL have been executed

▶ Statement Coverage

Spans multiple lines, more precise

▶ FSM Coverage

Show which state can be reached

▶ Toggle

Each Signal in gate-level netlist

▶ Function coverage

▶ Assertion

Successful or Failure

Suspicious Parts

- If one of the assertions fails, the IP is assumed untrusted.
- If coverage is not 100%, uncovered parts of the code (RTL, netlist) are assumed suspicious.

IC (System) Trust

- **Objective:**

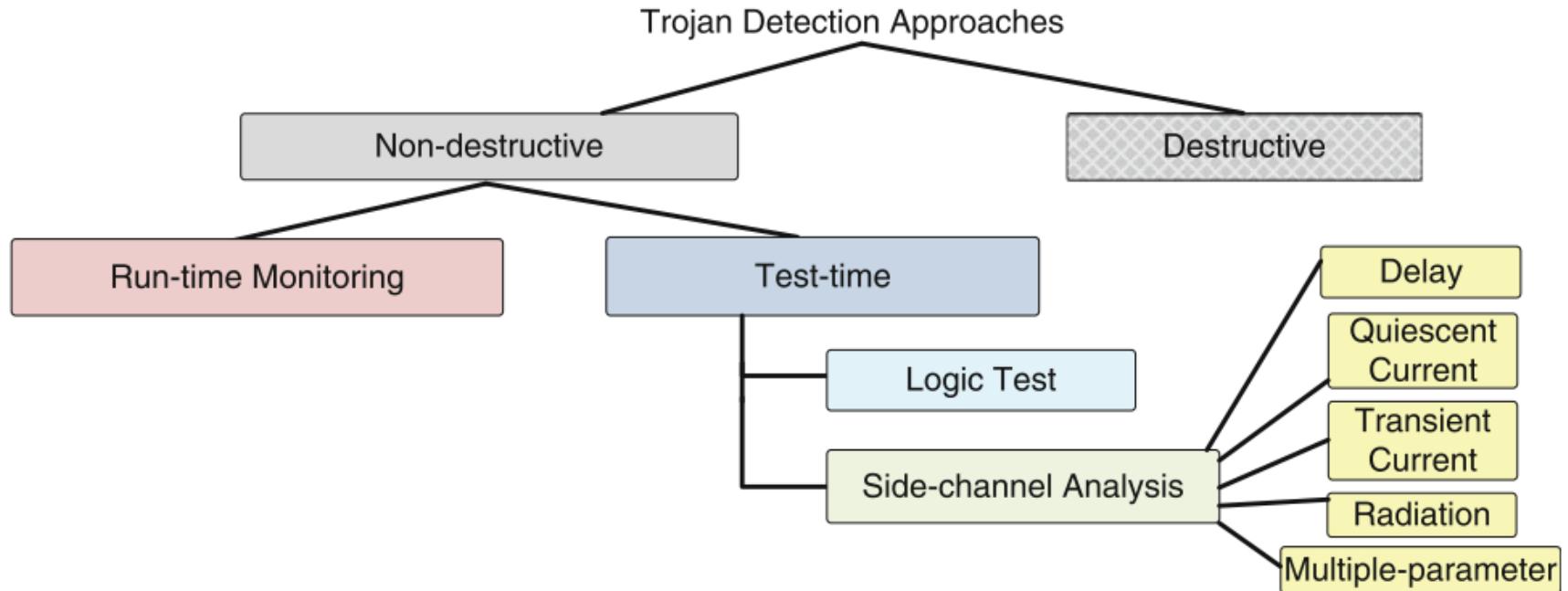
- Ensure that the *fabricated chip/system* will carry out only our desired function and nothing more.

- **Challenges:**

- **Tiny:** several gates to millions of gates
- **Quiet:** hard-to-activate (rare event) or triggered itself (time-bomb)
- **Hard to model:** human intelligence
- Conventional test and validation approaches fail to reliably detect hardware Trojans.
 - Focus on manufacture defects and does not target detection of additional functionality in a design



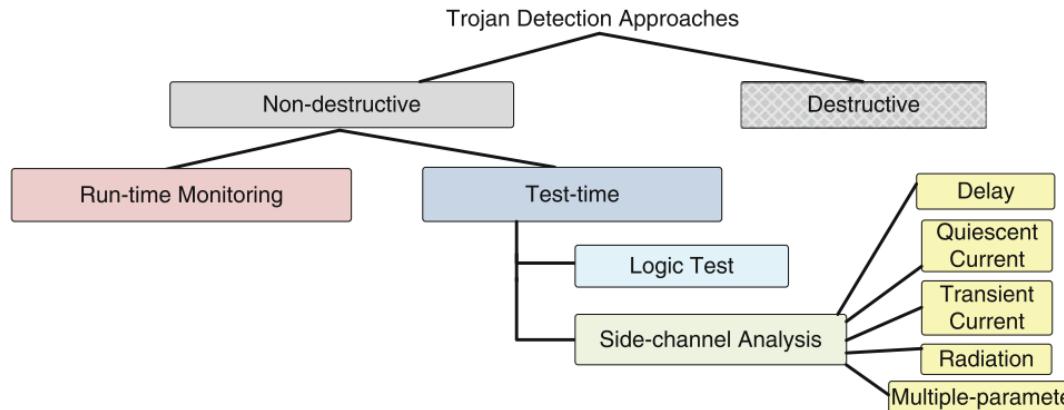
Classification of Trojan Detection Approaches



- **Destructive Approach:** Expensive and time consuming
 - Reverse engineering to extract layer-by-layer images by using delayering and Scanning Electron Microscope
 - Identify transistors, gates and routing elements by using a templatematching approach – **needs golden IC/layout**

Classification of Trojan Detection Approaches

- **Non-destructive Approach**
 - **Run-time monitoring:** Monitor abnormal behavior during run-time
 - Exploit pre-existing redundancy in the circuit
 - Compare results and select a trusted part to avoid an infected part of the circuit.
- **Test-time Authentication:** Detect Trojans throughout test duration.
 - Logic-testing-based approaches
 - Side-channel analysis-based approaches



Hardware Trojan Benchmarks

- A set of **trust benchmarks** for researchers in academia, industry, and government is needed to
 - Provide a baseline for examining diverse methods developed
 - Establishing a sound basis for the hardness of each benchmark instance
 - Help increase reproducibility of results by others who intend to employ certain methodologies in their design flow
- See NSF supported **Trust-Hub** website (www.trust-hub.org)
 - Complete taxonomy of Trojans
 - More than 120 trust benchmarks available which were designed at different abstraction levels, triggered in several ways, and have different effect mechanisms

Logic Testing Approach

- **Logic-testing approach** focuses on test-vector generation for
 - Activating a Trojan circuit
 - Observing its malicious effect on the payload at the primary outputs
 - Both functional and structural test vectors are applicable.
- **Pros:**
 - Straight-forward and easy to differentiate
- **Cons:**
 - The difficulty in exciting or observing low controllability or low observability nodes.
 - Intentionally inserted Trojans are triggered under rare conditions. (e.g., sequential Trojans)
 - It cannot trigger Trojans that are activated externally and can only observe functional Trojans.

Functional Test Deficiency

- Functional patterns could potentially detect a “functional” Trojan.
 - Exhaustive test would be effective, but certainly not applicable for large circuits
 - E.g. 64 input adder -> 2^{65} input combination (including carry in)
 - $2^{65} > 10^{18}$ – This is impractical
 - 100MHz is used -> 10^{10} s -> 317 years
 - Only a few and more effective patterns are used -> Trojan can escape.
 - The fault coverage is low for manufacturing test
- In practice, structural tests are used.

Functional Testing

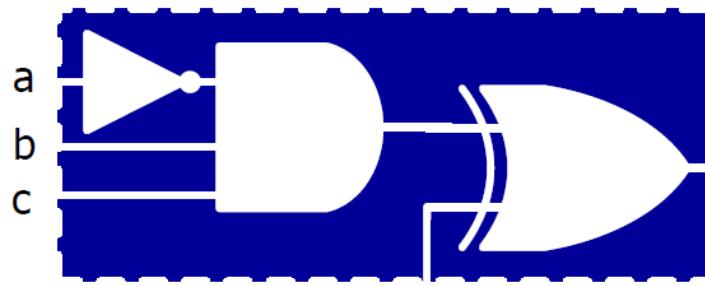
Feasible Trojan space inordinately large!

Deterministic test generation infeasible

A statistical approach is, more effective

- MERO: A Statistical Approach

- Find the rare events in the circuit
- Generate vectors to trigger each rare node **N times**
- Provides high confidence in detecting unknown Trojans!



Trojan Trigger Condition

$$a=0, b=1, c=1$$

From original circuit

MERO

- **MERO:**

- Generates a set of test vectors that can trigger each rare node to its rare value multiple times (N times)
- It improves the probability of triggering a Trojan activated by a rare combination of a selection of the nodes

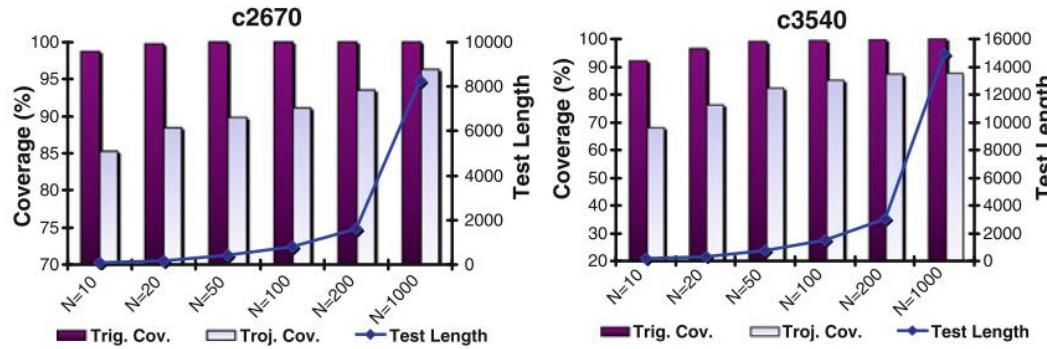
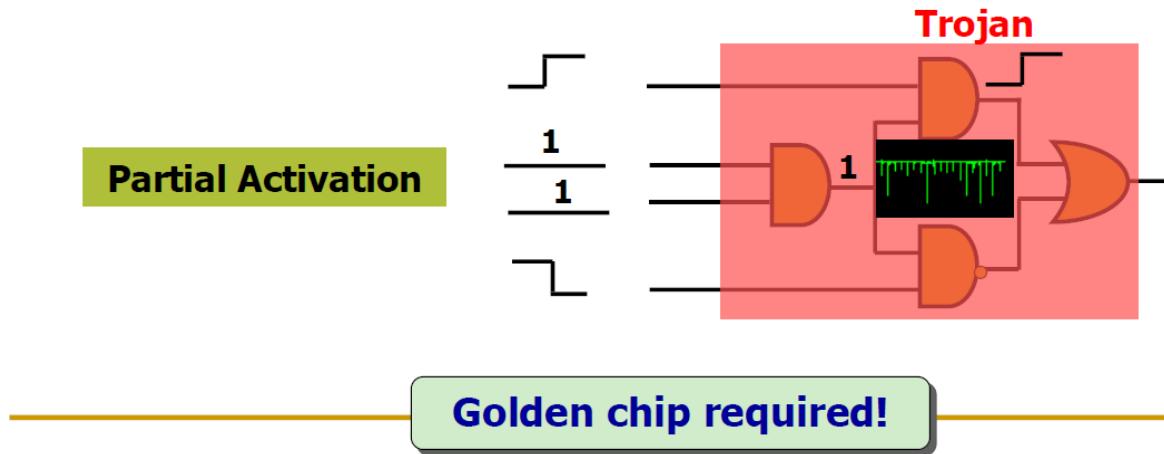


Fig. 15.6 Trigger coverage and Trojan coverage and test length for two ISCAS-85 benchmark circuits for different values of “ N ,” using the MERO approach [8]

- **Challenge:** Triggering each net N times in a large circuit is challenging

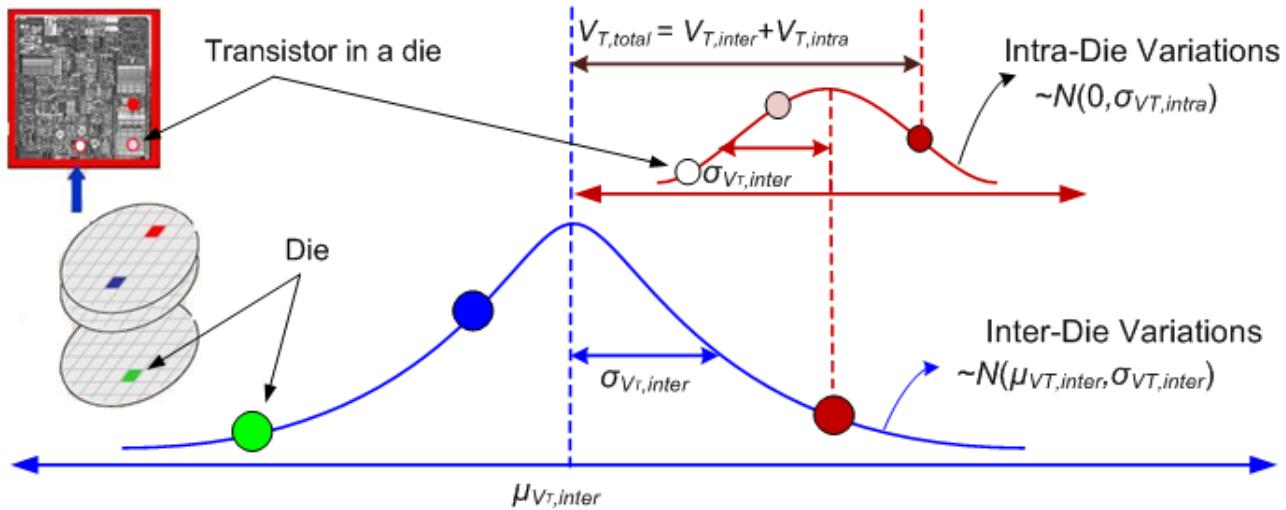
Side Channel Signal Analysis -- Power

- Hardware Trojans inserted in a chip can change the power consumption characteristics.
- **Partial activation** of Trojan can be extremely valuable for power analysis.
- The more number of cells in Trojan is activated the more the Trojan will draw current from power grid.



Side-channel Trojan Detection

- Side-Channel Approach for Trojan Detection relies on observing Trojan effect in physical side-channel parameter, such as **switching current, leakage current, path delay, electromagnetic (EM) emission**
 - Due to **process variations**, it is extremely challenging to detect the Trojan by considering F_{max} or I_{DDT} individually.



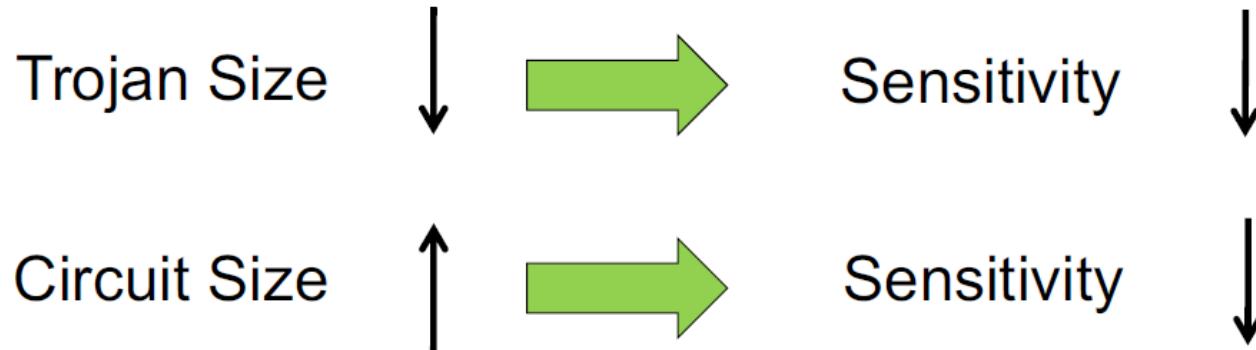
Side-channel signals

- All the side-channel analyses are based on observing the effect of an inserted Trojan on a physical parameter such as
 - **I_{DDQ}**: Extra gates will consume leakage power.
 - **I_{DDT}**: Extra switching activities will consume more dynamic power.
 - **Path Delay**: Additional gates and capacitance will increase path delay.
 - **EM**: Electromagnetic radiation due to switching activity
- **Pros & Cons**
 - **Pros**: It is effective for Trojan which does not cause observable malfunction in the circuits.
 - **Cons**: Large process variations in modern nanometer technologies and measurement noise can mask the effect of the Trojan circuits, especially for small Trojan.

Golden chip required!

Sensitivity Metric

Improving Detection Sensitivity



$$Sensitivity = \frac{I_{tampered} - I_{original}}{I_{original}} \times 100\%$$

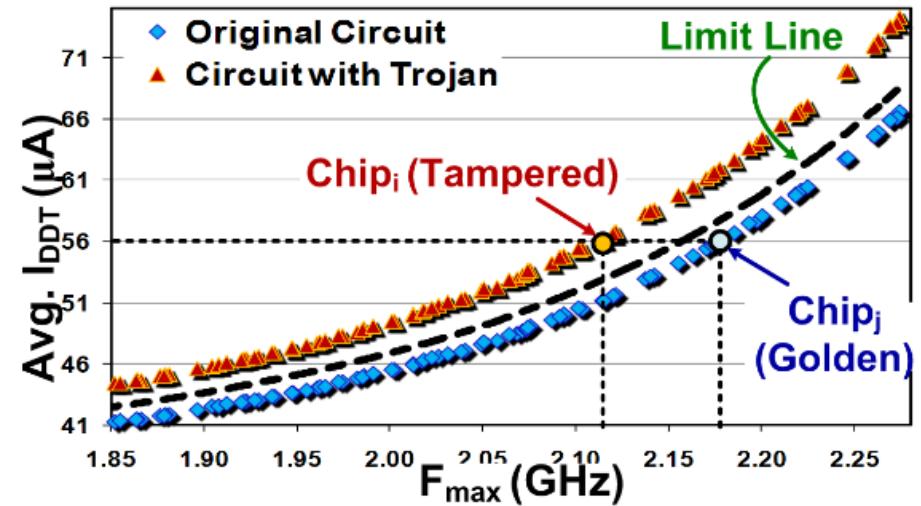
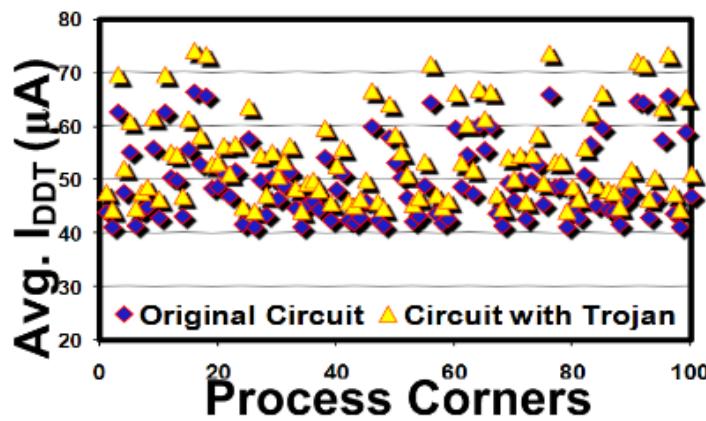
Comparing Approaches

	Logic Testing	Side-Channel Analysis
Pros	<ul style="list-style-type: none">• Robust under process noise• Effective for ultra-small Trojans	<ul style="list-style-type: none">• Effective for large Trojans• Easy to generate test vectors
Cons	<ul style="list-style-type: none">• Difficult to generate test vectors• Large Trojan detection challenging	<ul style="list-style-type: none">• Vulnerable to process noise• Ultra-small Trojan Det. challenging

- A combination of logic testing & side-channel analysis could provide the good coverage!
- Online validation approaches can potentially provide a second layer of defense!

Side channel Approach

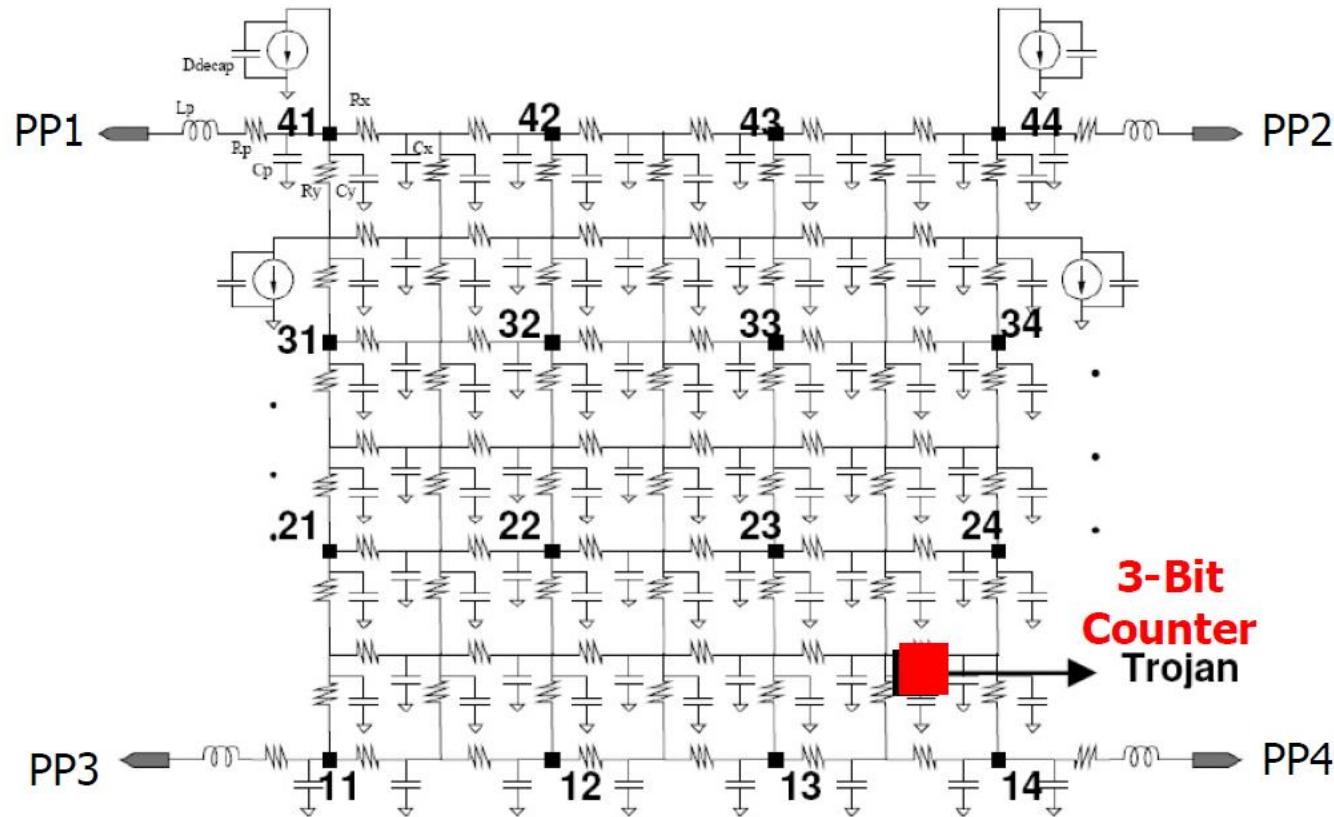
- *Multiple-parameter Trojan Detection*
 - Due to process variations, Trojan detection by F_{max} or I_{DDT} alone is challenging!



- Consider the intrinsic relationship between I_{DDT} and F_{max}

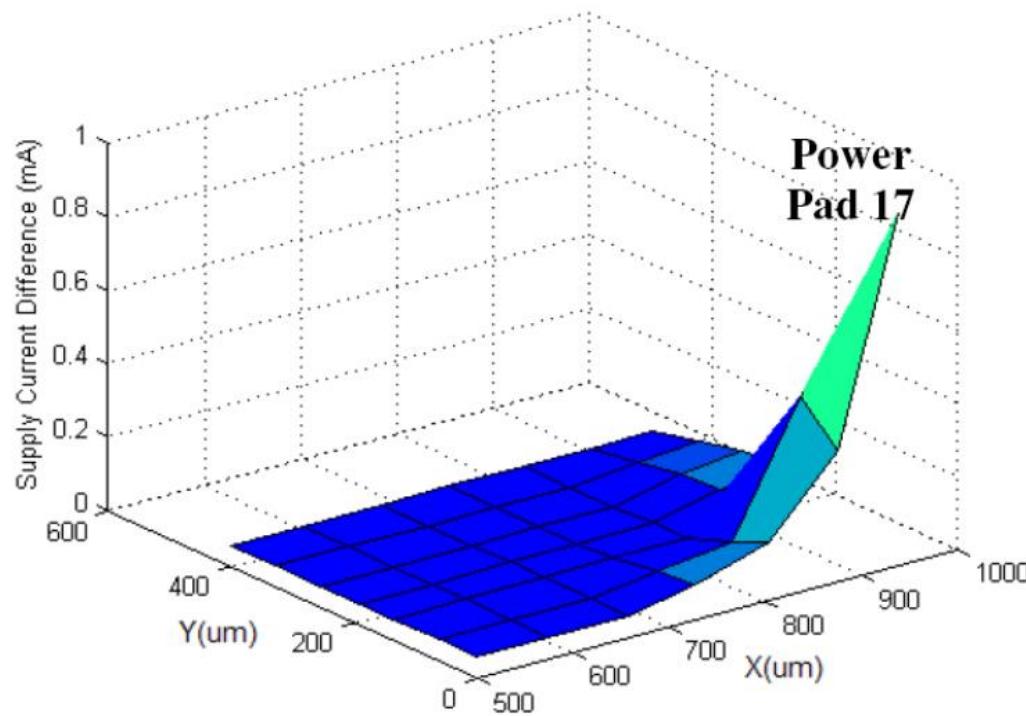
Golden chip required!

Trojan Inserted into s38417 Benchmark



PP: Power Pad

Power Analysis --Locality



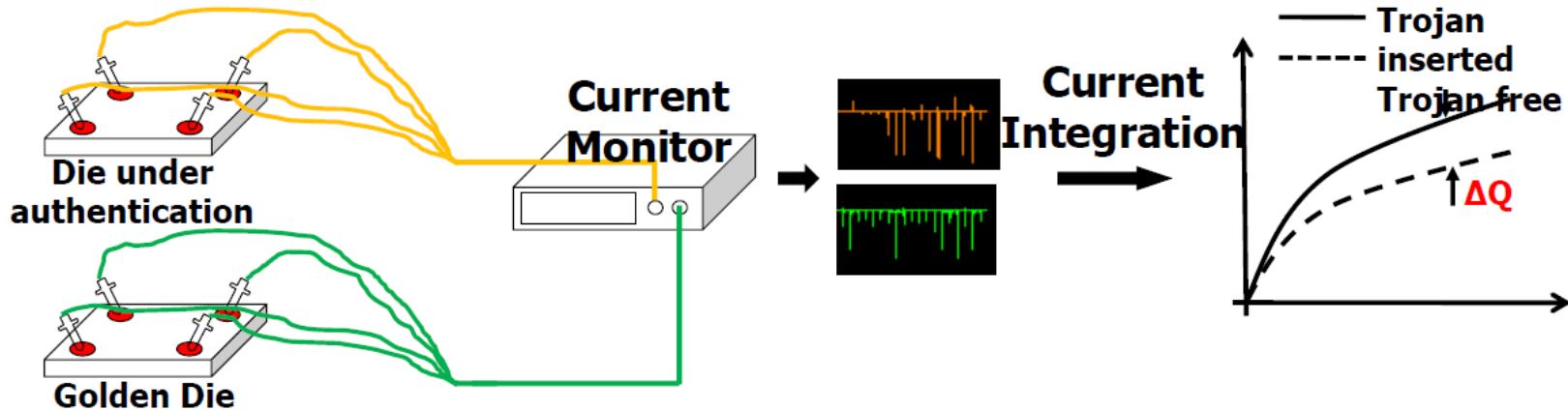
- Current difference measured from power pad 17 (Trojan-free vs Trojan-inserted)
- There is no change in layout of the circuit. Trojan was inserted in an unused space in the circuit layout.

Current (Charge) Integration Method

- Current consumption of Trojan-free and Trojan-inserted

$$Q_{trojan-free}(t) = \int I_{trojan_free}(t) \cdot dt$$

$$Q_{trojan-inserted}(t) = \int I_{trojan_inserted}(t) \cdot dt = \int (I_{trojan_free}(t) + I_{trojan}(t)) \cdot dt$$



Power Analysis -- Challenges

▶ Pattern Generation

- ▶ How to increase switching activity in Trojans?
- ▶ How to reduce background noise?
- ▶ Switching locality
- ▶ Random Patterns
 - ▶ No observation is necessary , Similar to test-per-clock

▶ Measurement Device Accuracy

- ▶ Measurement noise ✓

▶ Process Variations ✓

- ▶ Calibration ✓

▶ On-Chip Measurement

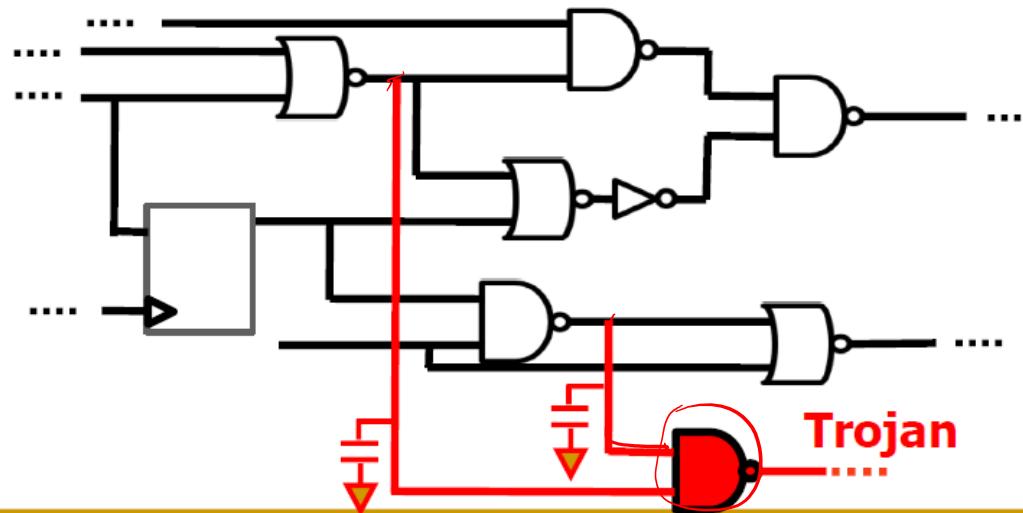
- ▶ Vulnerable to attack

▶ Authentication Time

- ▶ Trojans can be inserted randomly

Side Channel Analysis --Delay

- Hard to detect using power analysis are:
 - Distributed Trojans
 - Hard-to-activate Trojans
- **Path delay:** A change in physical dimension of the wires and transistors can also change path delay.
- We are developing new methods that can detect additional delays on each path of the circuit.

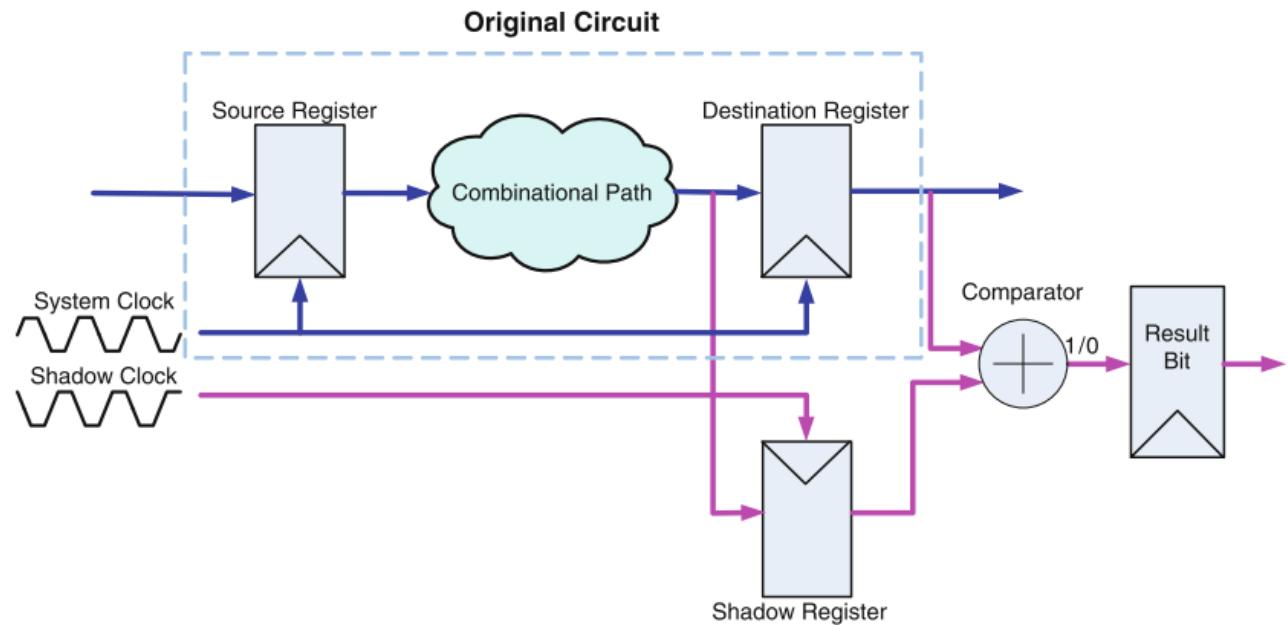


Delay Based Methods

- Shadow-register provides a possible solution for measuring internal path delay.
- From this architecture, it can be seen that the basic unit contains one shadow register, one comparator and one result register.

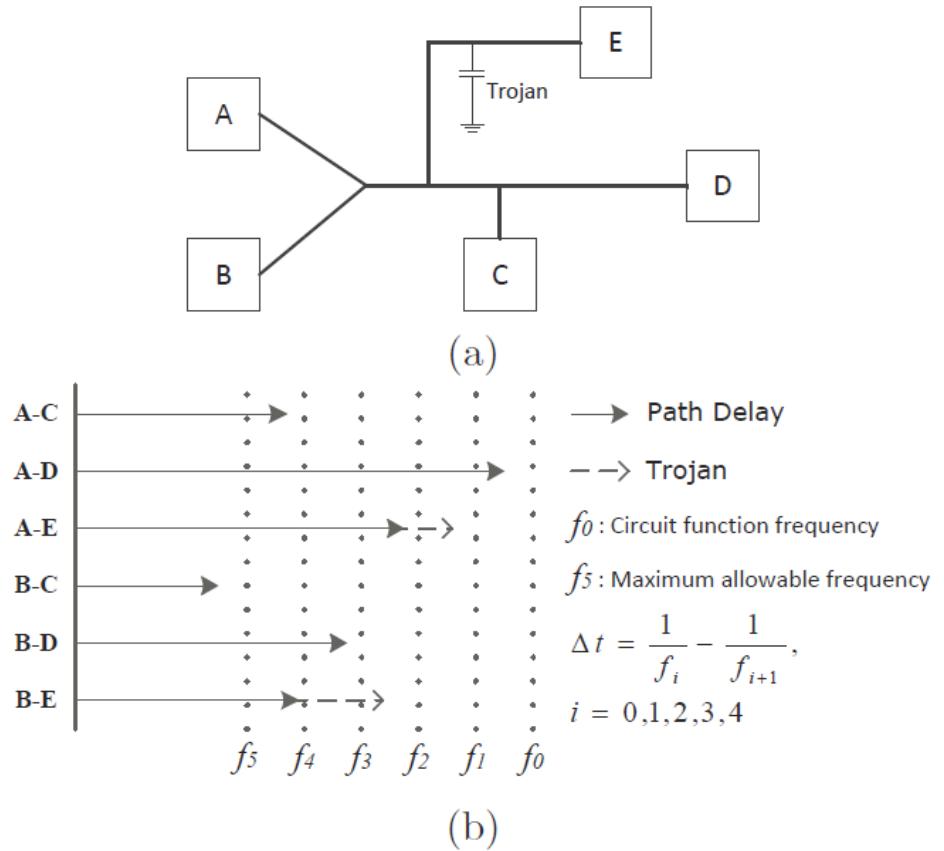
■ Limitations:

- PV
- Overhead
- S-clock
- Output



Clock Sweeping Technique

- Clock sweeping involves applying a pattern at different clock frequencies, from a lower speed to higher speeds.
- Some paths sensitized by the pattern which are longer than the current period start to fail when the clock speed increases.
- The obtained start-to-fail clock frequency can indicate the delays of the paths sensitized by the patterns



Delay Analysis – Challenges

- ▶ Major advantage over power analysis:
No activation is required.

- ▶ **Detection and Isolation**

- ▶ How significant is the delay inserted by Trojan?
- ▶ It depends on Trojan size and type
- ▶ Location: on short paths or long paths

- ▶ **Pattern Generation**

- ▶ Delay test patterns
- ▶ Path Coverage

- ▶ **Process Variations (V_{th} , L , T_{ox})**

- ▶ Impact circuit delay characteristics significantly
- ▶ Differentiate between Trojan and PV

- ▶ **Trojan can have impact on multiple paths (an advantage over PV)**

Trojan Detection

Trojan			Power Analysis	Delay Analysis	Fully Activation
Trojan Classification	Physical Characteristics	Type	Functional	D	P
		Parametric	P	D	P
		Size	Small	D	P
		Large	D	P	P
		Distribution	Tight	D	P
		Loose	P	D	P
		Structure	Modify Layout	P	D
	Activation Characteristics	Always-on			D
		Condition-based	Logic-based	D	P
		Sensor-based	D		
	Action Characteristics	Modify Function		D	P
		Modify Spec.	Defects	P	D
		Reliability	P	P	P

P: Detection is possible

D: High level of confidence

Design for Hardware Trust

- Since detecting Trojan is extremely challenging, design for hardware trust approaches are proposed to
 - **Improve hardware Trojan detection methods**
 - Improve sensitivity to power and delay
 - Rare event removal
 - **Prevent hardware Trojan insertion**
 - Design obfuscation

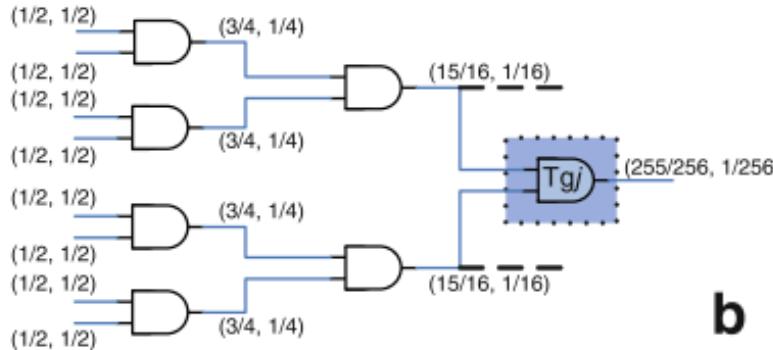
Rare Event Removal

- Intelligent attackers will choose low-frequency events to trigger the inserted Trojans.
- Improving controllability or observability can make rare events scarce, thereby facilitating detecting Trojans inside the design.
 - Design for Trojan test: inserting probing points
 - Inserting dummy scan flip-flops

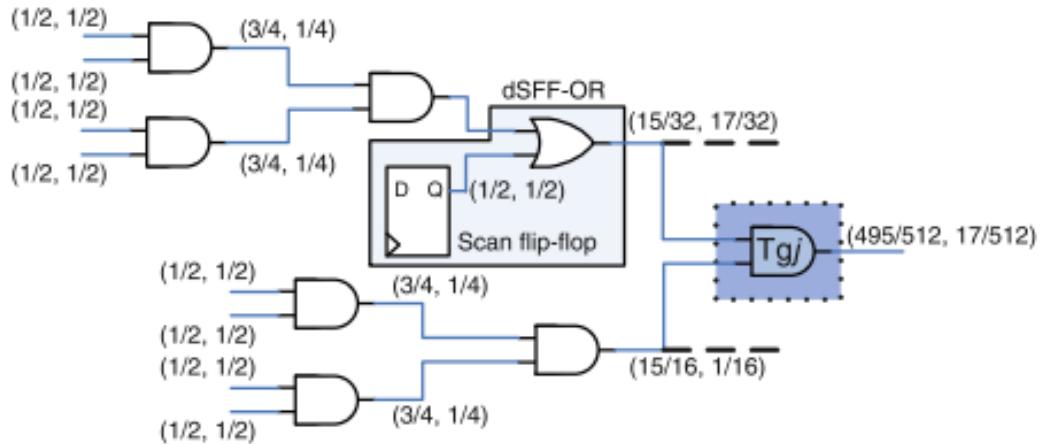
Increasing Probability of Partial/Full Activation

- Inserting dummy FFs on path with very low activation probability

a



b



Increasing Probability of Partial/Full Activation

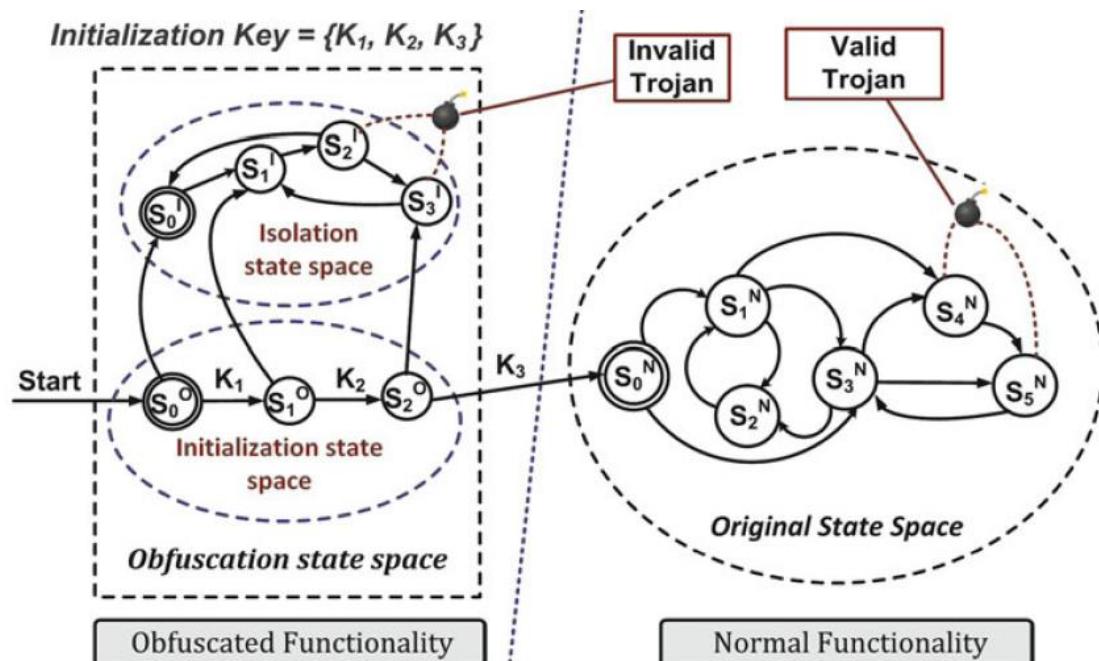
- Dummy scan flip-flops are inserted to control hard to-excite nodes.
- Usage:
 - **Full activation**: increase controllability
 - **Power-based**: generate switching activities
 - **Delay-based**: activate more paths to improve coverage

Trojan Prevention-Design obfuscation

- The objective is deterring attackers from inserting Trojans inside the design.
- Design obfuscation means that a design will be transformed to another one which is functionally equivalent to the original, but in which it is much harder for attackers to obtain complete understanding of the internal logic, making reverse engineering much more difficult to perform.
- It obfuscates the state transition function to add an obfuscated mode on top of the original functionality (called normal mode).

Design Obfuscation

- Specified pattern is able to guide the circuit into its normal mode.
- The transition arc K_3 is the only way the design can enter normal operation mode from the obfuscated mode.



ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 19:

Spectre and Meltdown Attacks

04/12/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Slide notes are based on: IEEE SPECTRUM article: "How the Spectre and Meltdown Hacks Really Worked"
by Nael Abu-Ghzaleh, Dmitry Ponomarev and Dmitry Evtyushkin

<https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked>

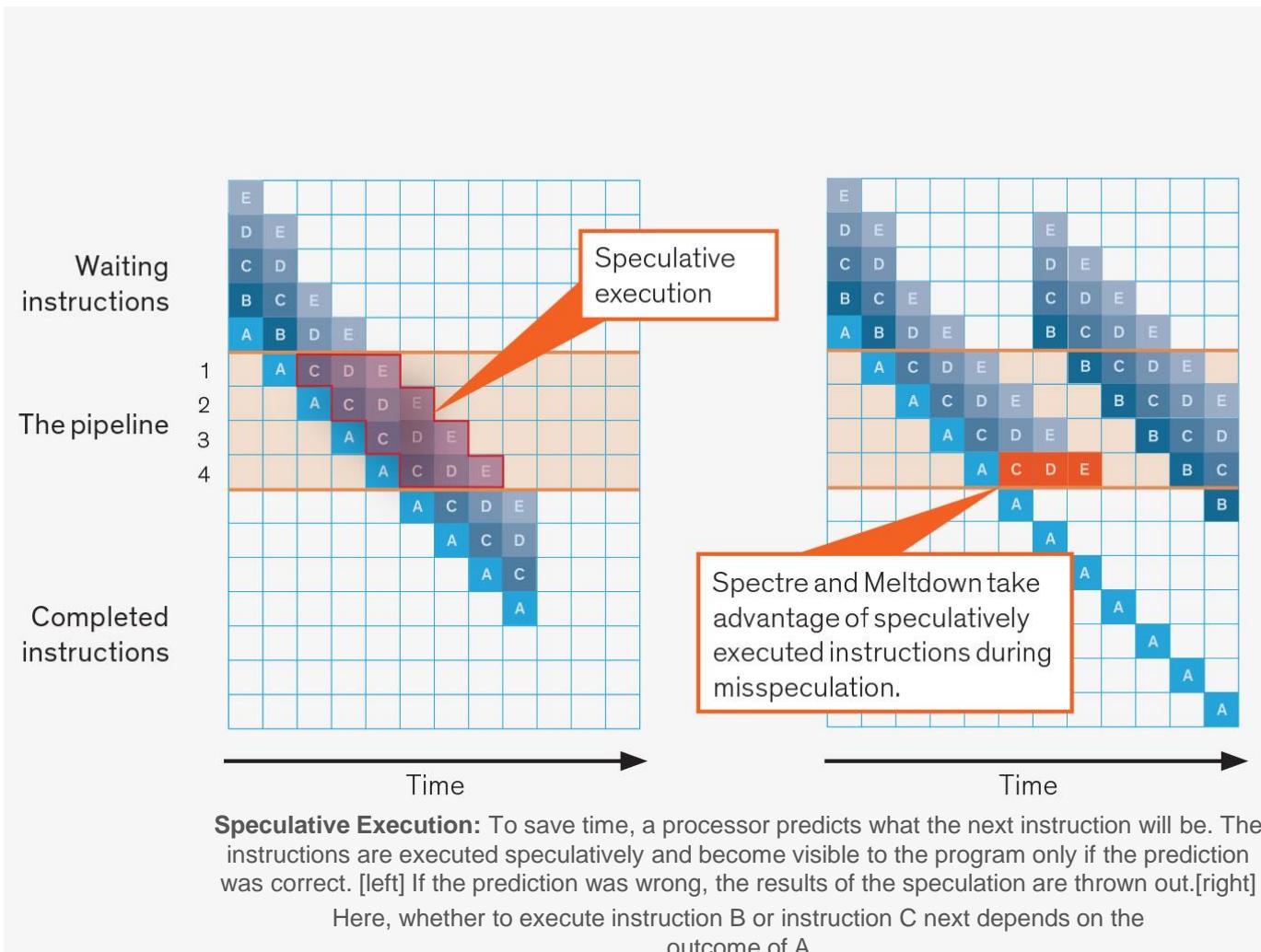
and

"Spectre Attacks: Exploiting Speculative Execution" by Paul Kocher et al.,

Modern CPU Vulnerabilities

- In 2017, several research groups including Cyberus Technology, Google Project Zero, Graz University of Technology, Rambus, University of Adelaide, and University of Pennsylvania announced major CPU vulnerabilities that took advantage of speculative execution.
- Meltdown could hack all Intel x86 microprocessors and IBM Power processors, as well as some ARM-based processors. Spectre and its many variations added Advanced Micro Devices (AMD) processors to that list. In other words, all modern CPUs were vulnerable.
- This required response from CPU manufacturers and also Operating Systems designers such as Microsoft, Apple and Linux.

Speculative Execution



Meltdown and Spectre Attacks

- Meltdown, Spectre, and their variants all follow the same pattern. First, they trigger speculation to execute code desired by the attacker. This code reads secret data without permission.
- Then, the attacks communicate the secret using **Flush and Reload** or a similar side channel. That last part is well understood and similar in all of the attack variations. Thus, the attacks differ only in the first component, which is how to trigger and exploit speculation.

Cache based side channel attacks

- **Flush and Reload**, begins with the attacker removing shared data from the cache using the “flush” instruction. The attacker then waits for the victim to access that data. Because it’s no longer in the cache, any data the victim requests must be brought in from main memory.
- Later, the attacker accesses the shared data while timing how long this takes. A cache hit—meaning the data is back in the cache—indicates that the victim accessed the data. A cache miss indicates that the data has not been accessed.
- Simply by measuring how long it took to access data, the attacker can determine which cache sets were accessed by the victim. This knowledge of which cache sets were accessed and which were not can lead to the discovery of encryption keys and other secrets

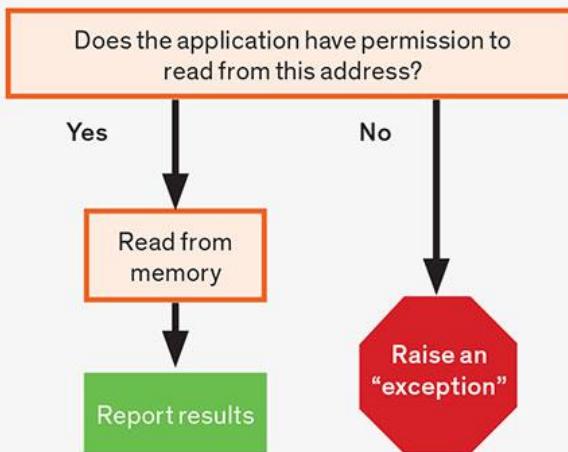
Meltdown Attacks

- Meltdown attacks **exploit speculation within a single instruction**. For example, memory-read operations are often dependent on the instruction satisfying the permissions associated with the memory address being read. An application usually has permission to read only from memory that's been assigned to it, not from memory allocated to the operating system or some other user's program. Logically, permissions need to be checked before allowing the read to proceed.
- However, Intel microprocessors read the memory location before checking permissions, but only "commit" the instruction—making the results visible to the program—when the permissions are satisfied. But because the secret data has been retrieved speculatively, it can be discovered using a side channel, making Intel processors vulnerable to this attack.

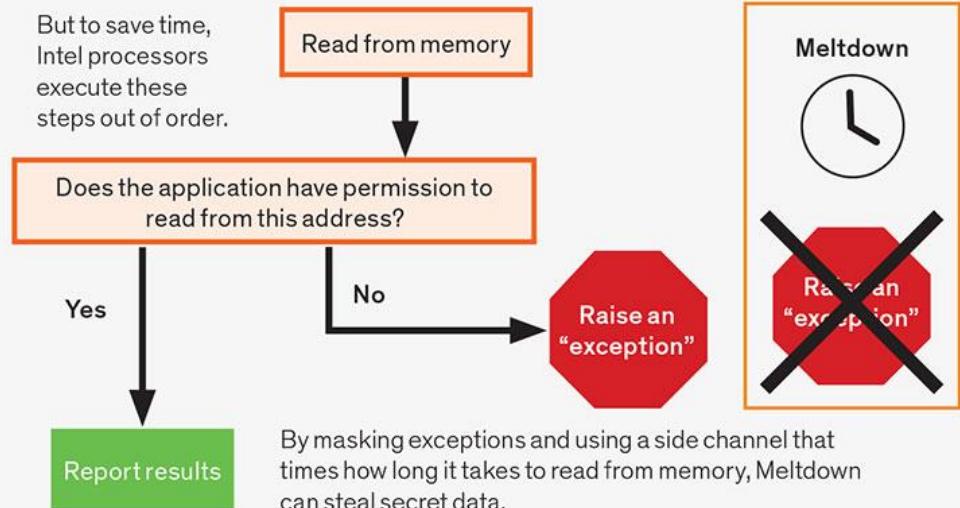
Meltdown Attacks

MELTDOWN

Logically, reading from memory should work like this:



But to save time, Intel processors execute these steps out of order.



By masking exceptions and using a side channel that times how long it takes to read from memory, Meltdown can steal secret data.

Spectre Attacks

- Spectre attacks manipulate the branch-prediction system. This system has three parts: the branch-direction predictor, the branch-target predictor (cache), and the return stack buffer.
- Each of these three structures can be exploited in different ways. The predictor can be deliberately mistrained.
- In this case, the attacker executes seemingly innocent code designed to befuddle the system. Later, the attacker deliberately executes a branch that will misspeculate, causing the program to jump to a piece of code chosen by the attacker, called a gadget. The gadget then sets about stealing data.

Spectre attack

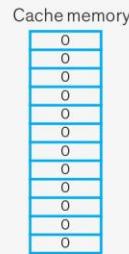
SPECTRE VARIANT 1

Spectre v1 can be summed up in the following piece of code:

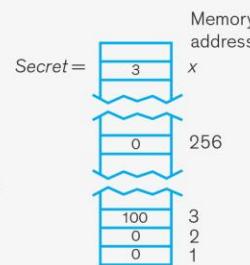
```
If(x<256){  
    secret=array1[x]  
    y=array2[secret]  
}
```

1. The code is run several times with x less than 256. This primes the branch predictor to expect x to be less than 256 the next time.

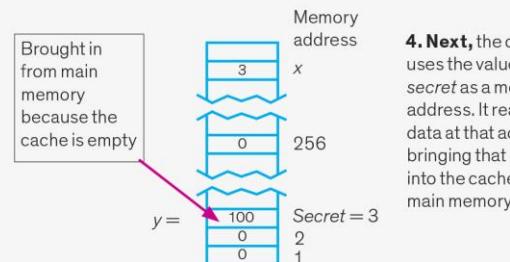
Take the branch?



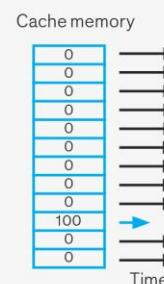
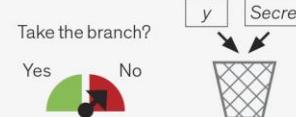
2. The attacker empties the processor's cache using flush instructions, so that any data read by the program must be brought in from main memory.



3. The attacker runs the code with x set to a value greater than 256. The processor begins to "speculatively" execute the rest of the code as if x were less than 256. The memory address at x contains secret data the attacker wants. The software assigns this data to the variable $secret$.



5. Eventually, the processor realizes that it should not have taken the branch. It does not make the values $secret$ and y visible to the program.



6. The attacker accesses each address in the cache systematically. Because the memory address equal to the secret is the only one whose data has already been brought in from main memory, it takes less time to access than all the others, thereby revealing the secret.

Protection against Meltdown and Spectre

- **Operating Systems:**

- One defense, called kernel page-table isolation (KPTI) is now built into Linux and other operating systems.
- KPTI and similar systems defend against Meltdown by making secret data in memory, such as the OS, inaccessible when a user's program (and potentially an attacker's program) is running. It does this by removing the forbidden parts from the page table. That way, even speculatively executed code cannot access the data.
- KVA Shadow trap for Windows: Privileged kernel-mode memory must not be mapped into the address space when untrusted user mode code runs

Protection against Meltdown and Spectre

- **Microcode changes:**
- Intel and AMD adjusted their microcode to change the behavior of some assembly-language instructions in ways that limit speculation. For example, Intel engineers added options that interfere with some of the attacks by allowing the operating system to empty the branch-predictor structures in certain circumstances.
- **New Architectures:**
- Proposals for new processor architectures that would introduce structures on the CPU that are dedicated to speculation and separate from the processor's cache and other hardware. This way, any operations that are executed speculatively but are not eventually committed are never visible. If the speculation result is confirmed, the speculative data is sent to the processor's main structures.

Spectre Attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- The code fragment begins with a bounds check on x which is essential for security. This check prevents the processor from reading sensitive memory outside of array1 . Otherwise, an out-of-bounds input x could trigger an exception or could cause the processor to access sensitive memory by supplying $x = (\text{address of a secret byte to read}) - (\text{base address of array1})$.
- During speculative execution, the conditional branch for the bounds check can follow the incorrect path. In this example, suppose an adversary causes the code to run such that:
 - the value of x is maliciously chosen (out-of-bounds), such that $\text{array1}[x]$ resolves to a secret byte k somewhere in the victim's memory;
 - array1_size and array2 are *uncached*, but k is cached; and
 - previous operations received values of x that were valid, leading the branch predictor to assume the if will likely be true.
- This cache configuration can occur naturally or can be created by an adversary, e.g., by causing eviction of array1_size and array2 then having the kernel use the secret key in a legitimate operation.

Spectre Attack Variant 1

- Figure 1 illustrates the four cases of the bounds check in combination with speculative execution. Before the result of the bounds check is known, the CPU speculatively executes code following the condition by predicting the most likely outcome of the comparison.

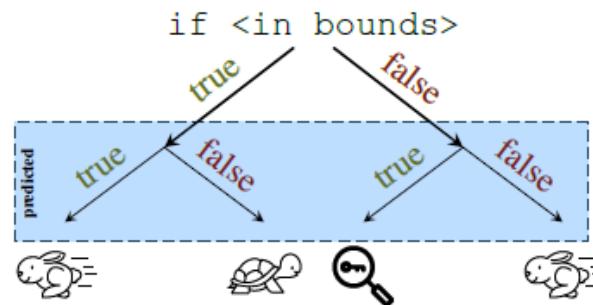


Fig. 1: Before the correct outcome of the bounds check is known, the branch predictor continues with the most likely branch target, leading to an overall execution speed-up if the outcome was correctly predicted. However, if the bounds check is incorrectly predicted as true, an attacker can leak secret information in certain scenarios.

Spectre Attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- When the compiled code above runs, the processor begins by comparing the malicious value of x against $array1_size$. Reading $array1_size$ results in a cache miss, and the processor faces a substantial delay until its value is available from DRAM.
- In the meantime, the branch predictor assumes the *if* will be true. Consequently, the speculative execution logic adds x to the base address of $array1$ and requests the data at the resulting address from the memory subsystem. This read is a cache hit, and quickly returns the value of the secret byte k .
- The speculative execution logic then uses k to compute the address of $array2[k * 4096]$. It then sends a request to read this address from memory (resulting in a cache miss). While the read from $array2$ is already in flight, the branch result may finally be determined. The processor realizes that its speculative execution was erroneous and rewinds its register state.
- However, the speculative read from $array2$ affects the cache state in an address-specific manner, where the address depends on k .

Spectre Attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- To complete the attack, the adversary measures which location in *array2* was brought into the cache, e.g., via Flush+Reload or Prime+Probe.
- This reveals the value of *k*, since the victim's speculative execution cached *array2[k*4096]*.

ECE 586 Hardware Security and Advanced Computer Architecture

LECTURE 20:

Multiprocessors and Thread-Level Parallelism

04/17/2023

Erdal Oruklu, PhD

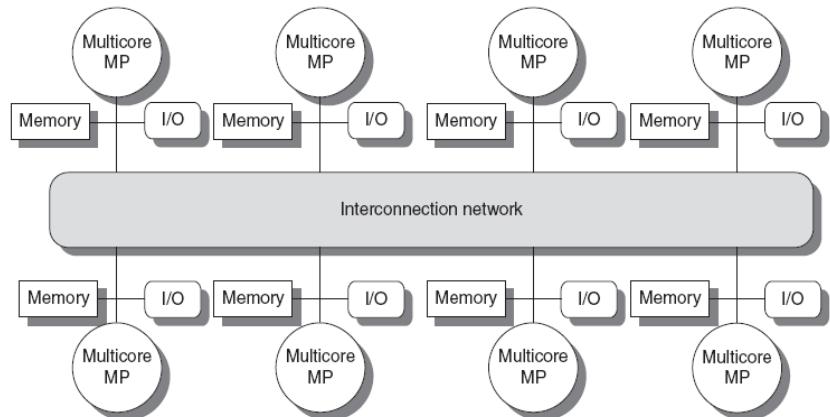
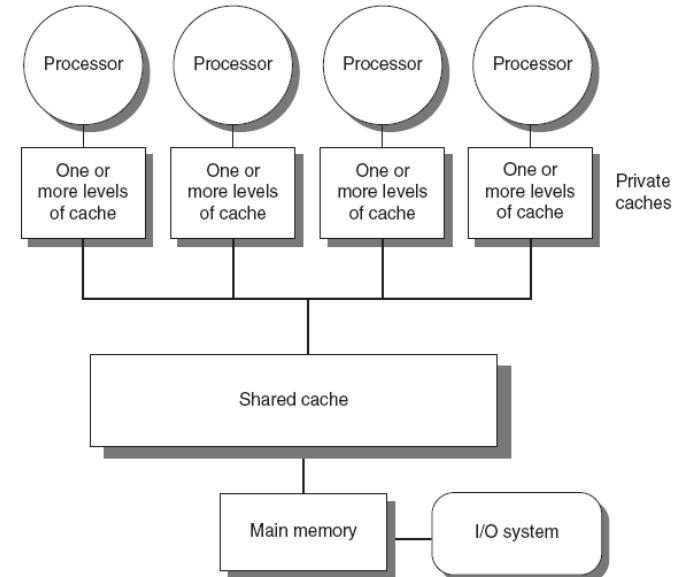
Illinois Institute of Technology
Department of Electrical and Computer Engineering

Introduction

- Thread-Level parallelism
 - Have multiple program counters
 - Uses MIMD model
 - Targeted for tightly-coupled shared-memory multiprocessors
- For n processors, need n threads
- Amount of computation assigned to each thread = grain size
 - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Types

- Symmetric multiprocessors (SMP)
 - Small number of cores
 - Share single memory with uniform memory latency
- Distributed shared memory (DSM)
 - Memory distributed among processors
 - Non-uniform memory access/latency (NUMA)
 - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks



Cache Coherence

- Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Cache Coherence

- Coherence
 - All reads by any processor must return the most recently written value
 - Writes to the same location by any two processors are seen in the same order by all processors
- Consistency
 - When a written value will be returned by a read
 - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence

- Coherent caches provide:
 - *Migration*: movement of data
 - *Replication*: multiple copies of data
- Cache coherence protocols
 - Directory based
 - Sharing status of each block kept in one location
 - Snooping
 - Each core tracks sharing status of each block

Snoopy Coherence Protocols

- Write invalidate
 - On write, invalidate all other copies
 - Use bus itself to serialize
 - Write cannot complete until bus access is obtained

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

- Write update
 - On write, update all copies

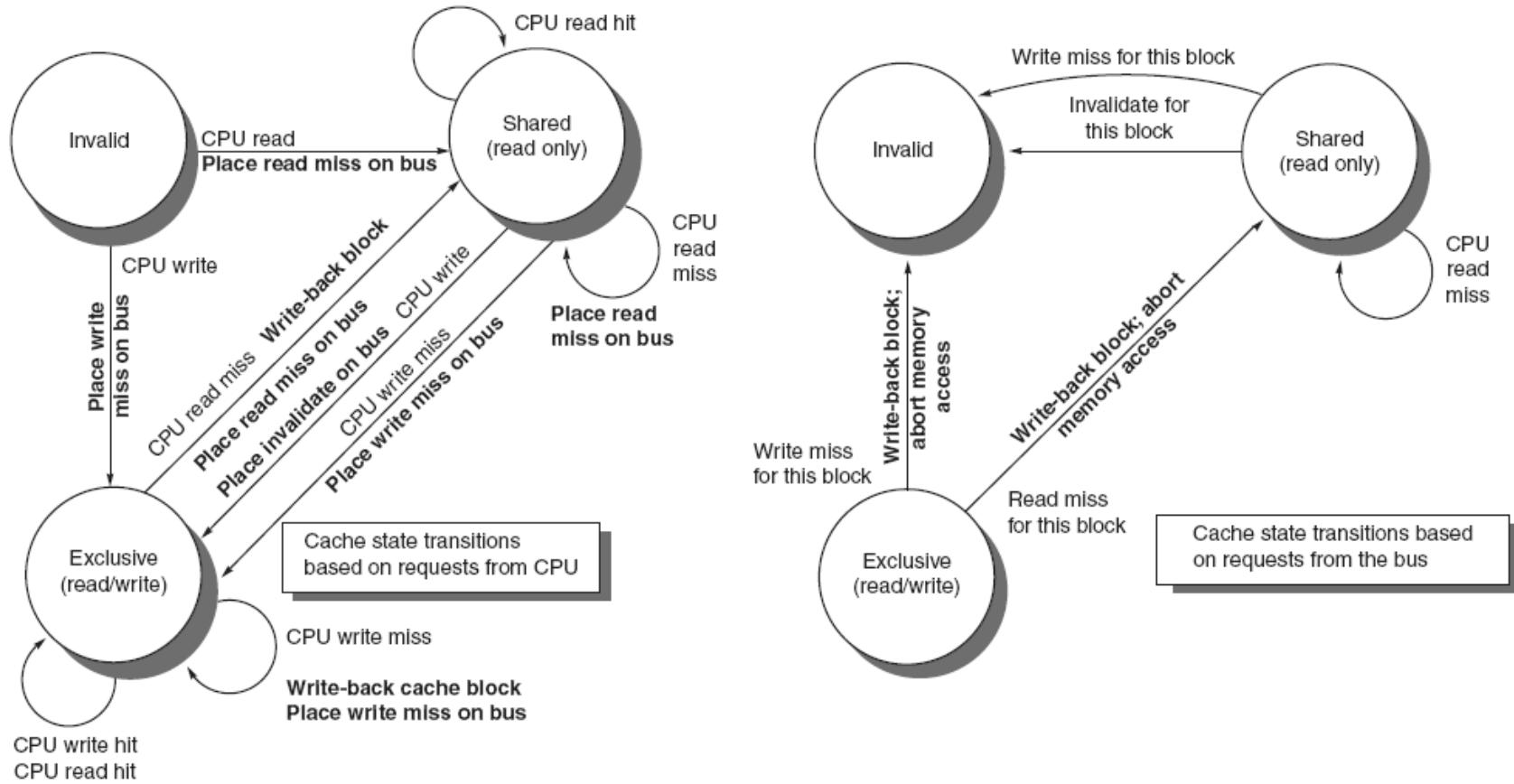
Snoopy Coherence Protocols

- Locating an item when a read miss occurs
 - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
 - Only writes to shared lines need an invalidate broadcast
 - After this, the line is marked as exclusive

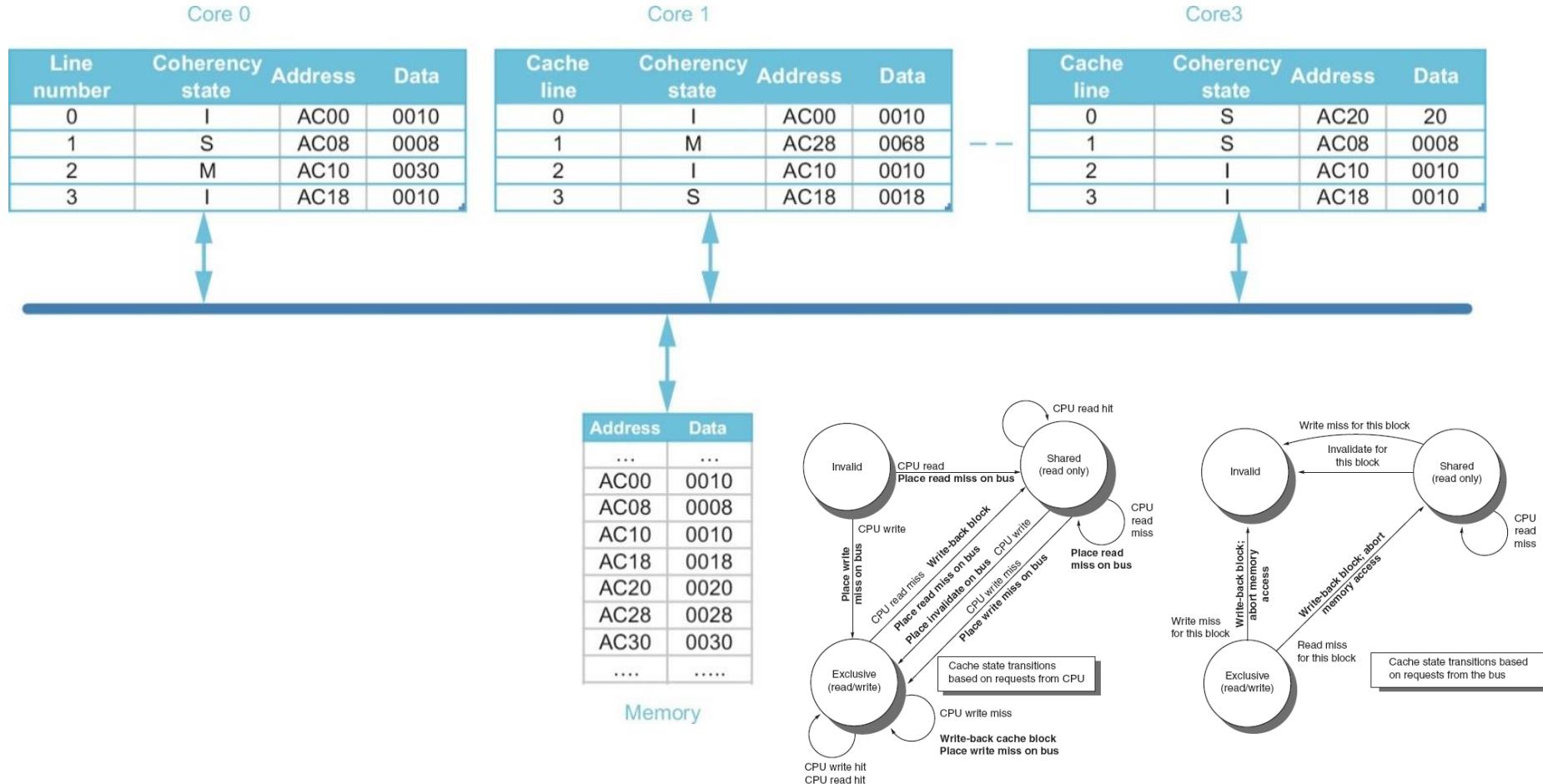
Snoopy Coherence Protocols

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

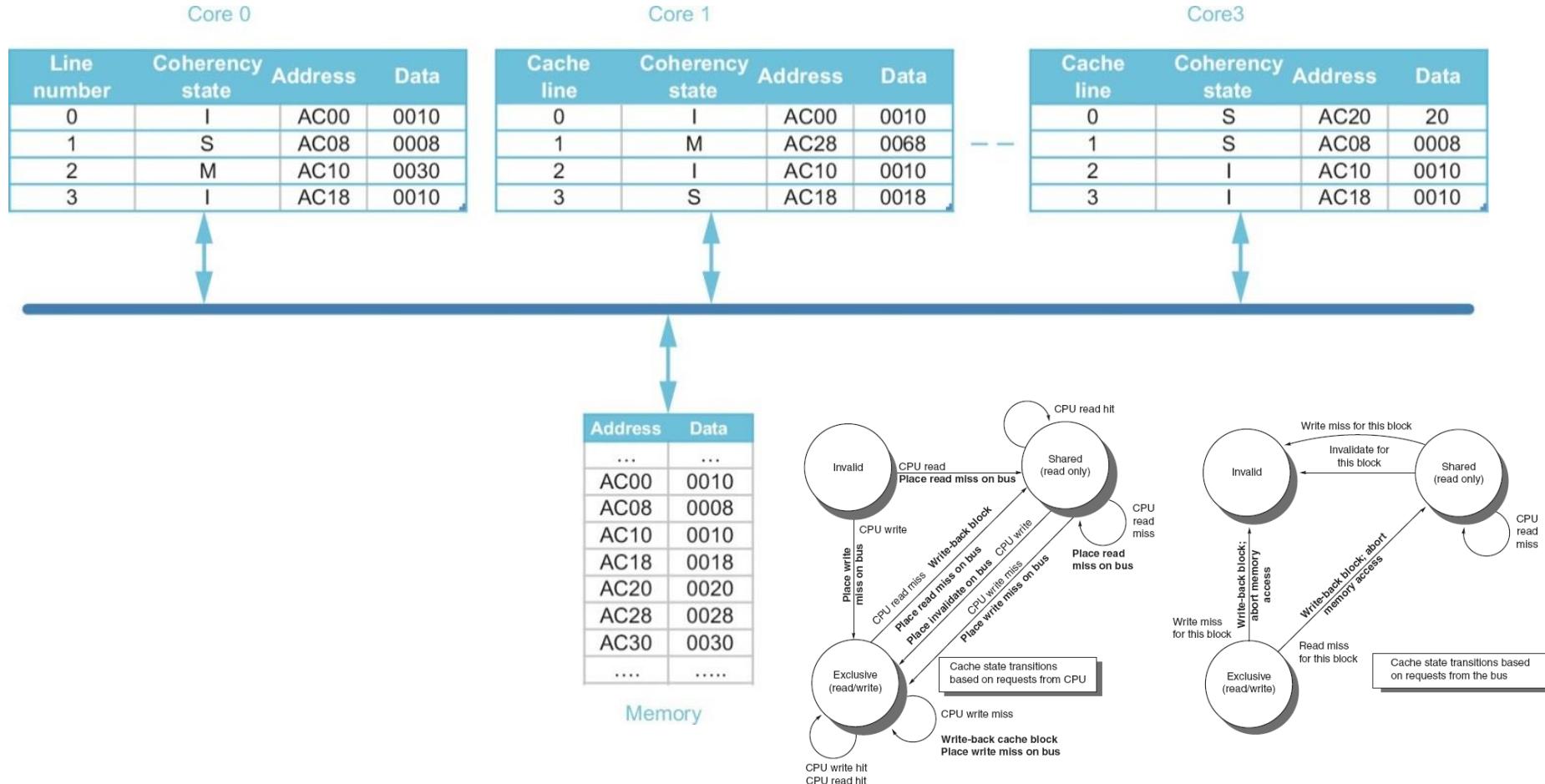
Snoopy Coherence Protocols



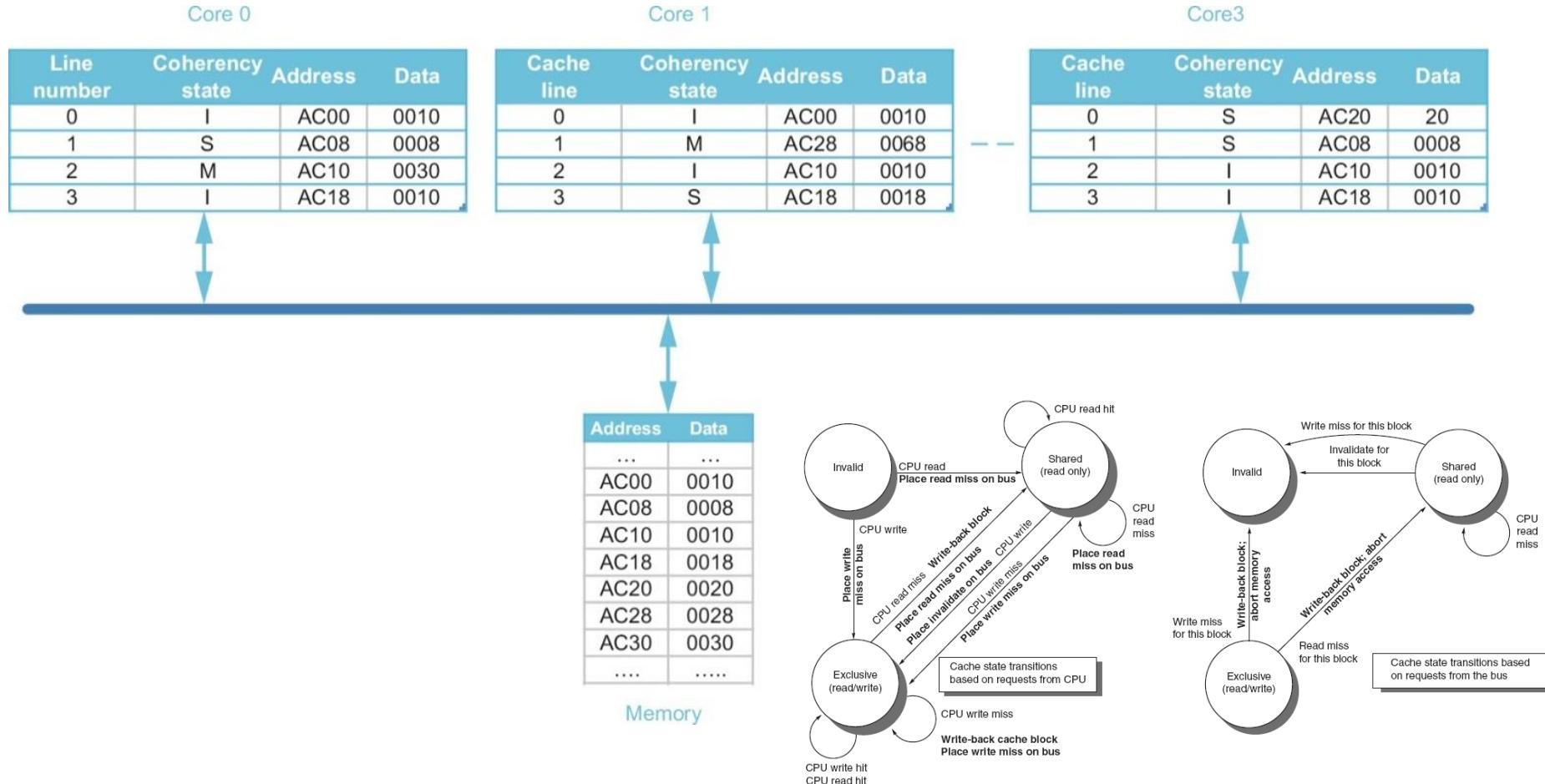
Example for Snooping Protocol (SMP)



Example for Snooping Protocol (SMP)



Example for Snooping Protocol (SMP)

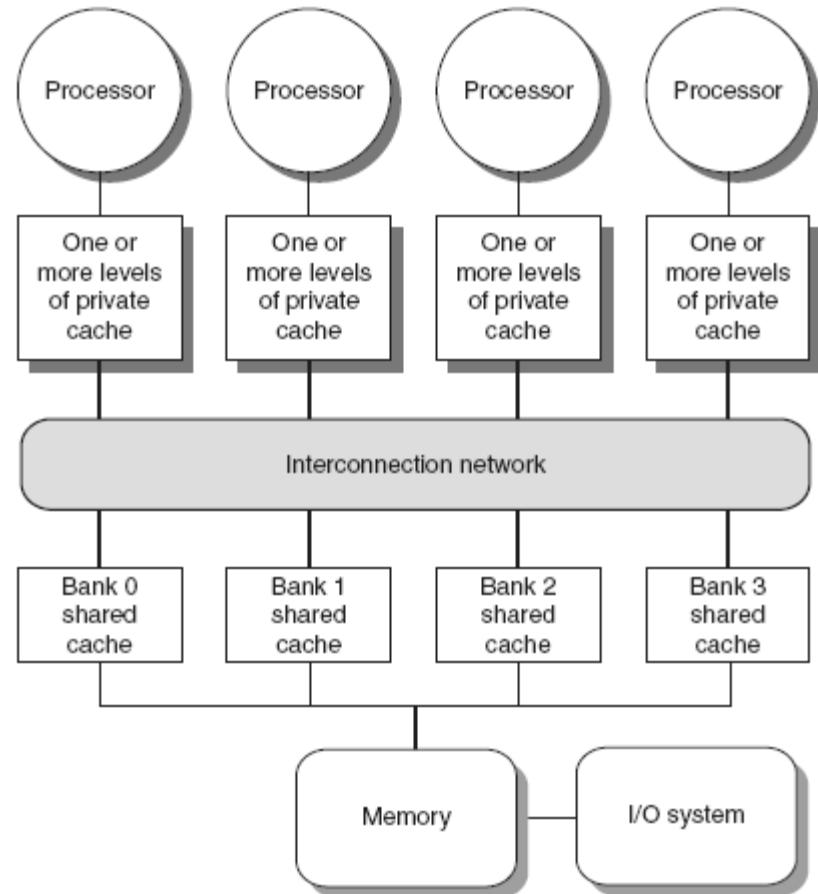


Snoopy Coherence Protocols

- Complications for the basic MSI protocol:
 - Operations are not atomic
 - E.g. detect miss, acquire bus, receive a response
 - Creates possibility of deadlock and races
 - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
 - Add exclusive state to indicate clean block in only one cache (MESI protocol)
 - Prevents needing to write invalidate on a write
 - Owned state

Coherence Protocols: Extensions

- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
 - Duplicating tags
 - Place directory in outermost cache
 - Use crossbars or point-to-point networks with banked memory



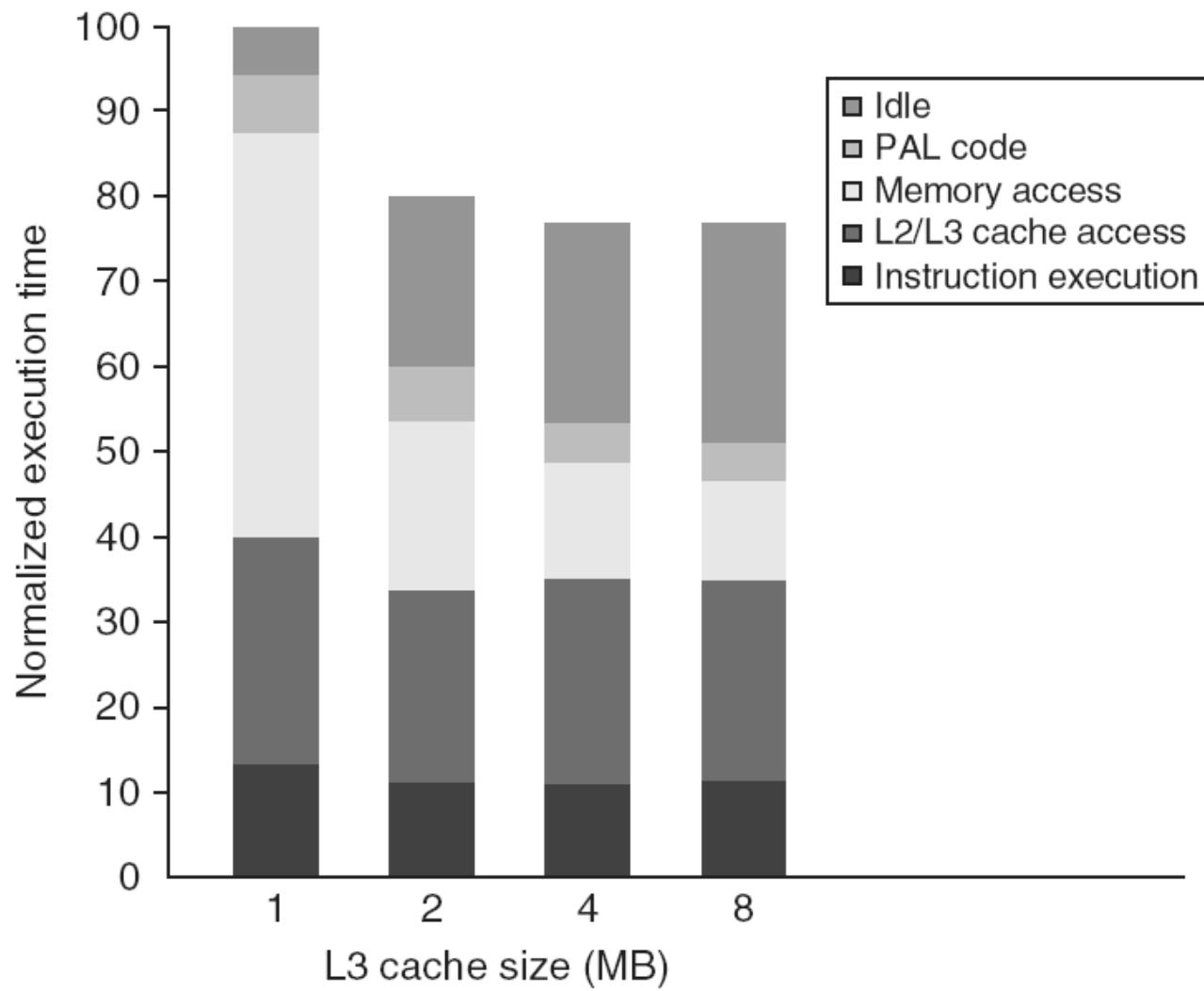
Coherence Protocols

- AMD Opteron:
 - Memory directly connected to each multicore chip in NUMA-like organization
 - Implement coherence protocol using point-to-point links
 - Use explicit acknowledgements to order operations

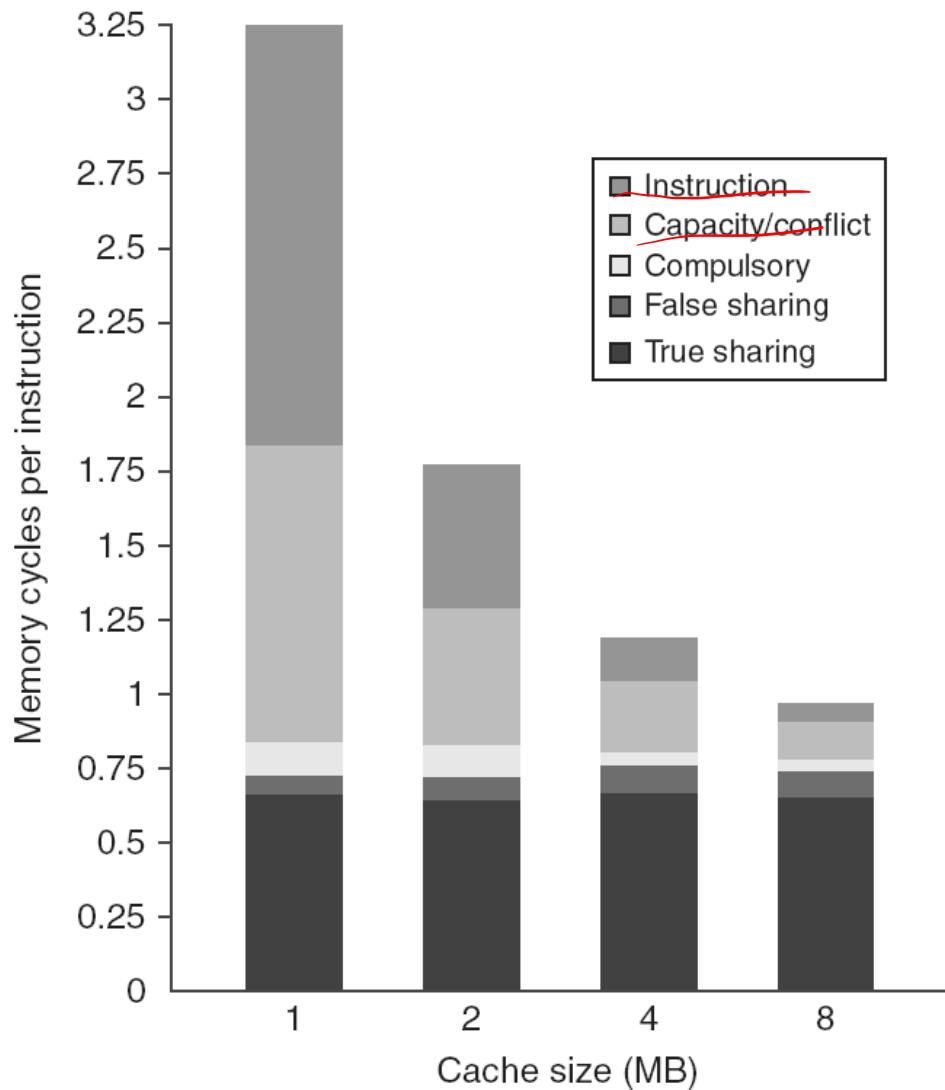
Performance

- Coherence influences cache miss rate
 - Coherence misses
 - True sharing misses
 - Write to shared block (transmission of invalidation)
 - Read an invalidated block
 - False sharing misses
 - Read an unmodified word in an invalidated block

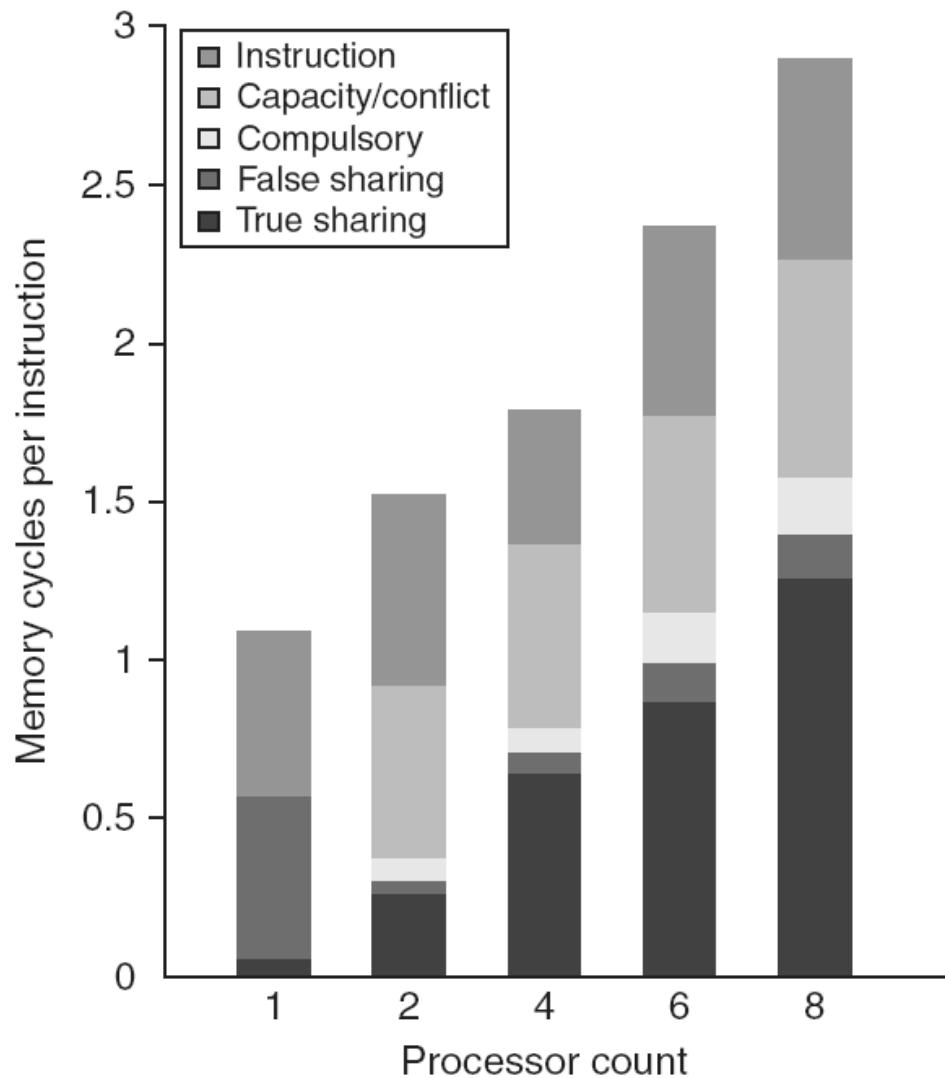
Performance Study: Commercial Workload



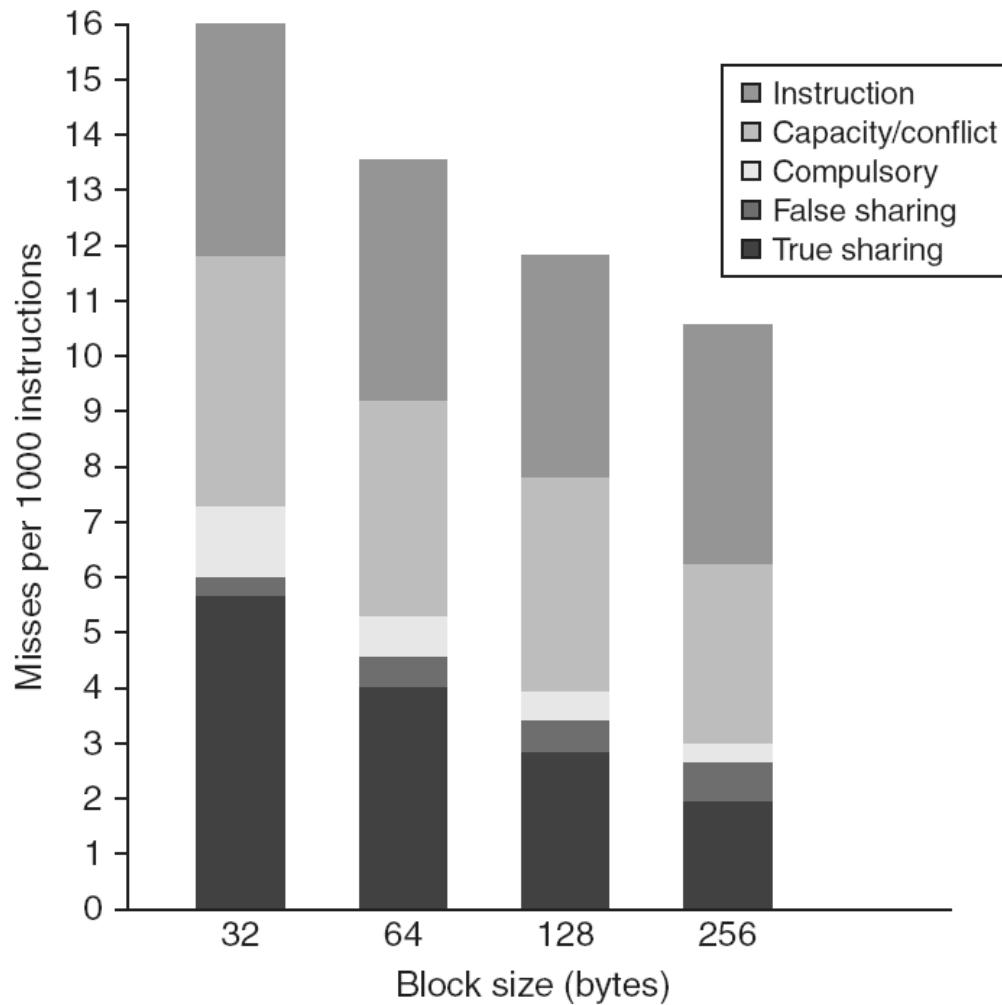
Performance Study: Commercial Workload



Performance Study: Commercial Workload

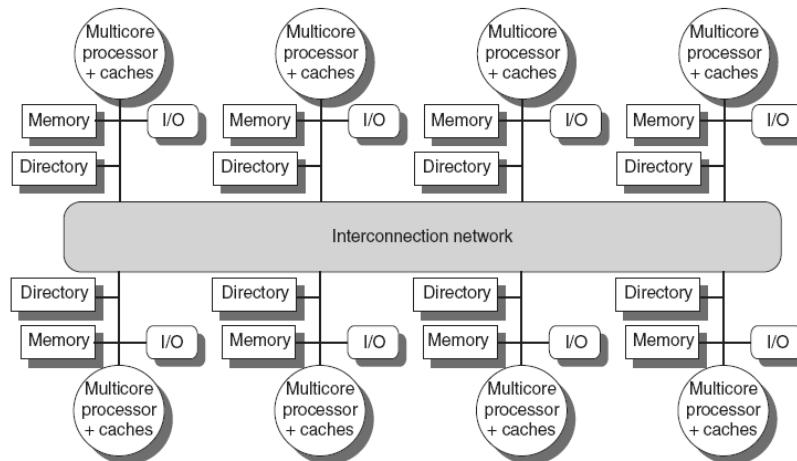


Performance Study: Commercial Workload



Directory Protocols

- Directory keeps track of every block
 - Which caches have each block
 - Dirty status of each block
- Implement in shared L3 cache
 - Keep bit vector of size = # cores for each block in L3
 - Not scalable beyond shared L3
- Implement in a distributed fashion:



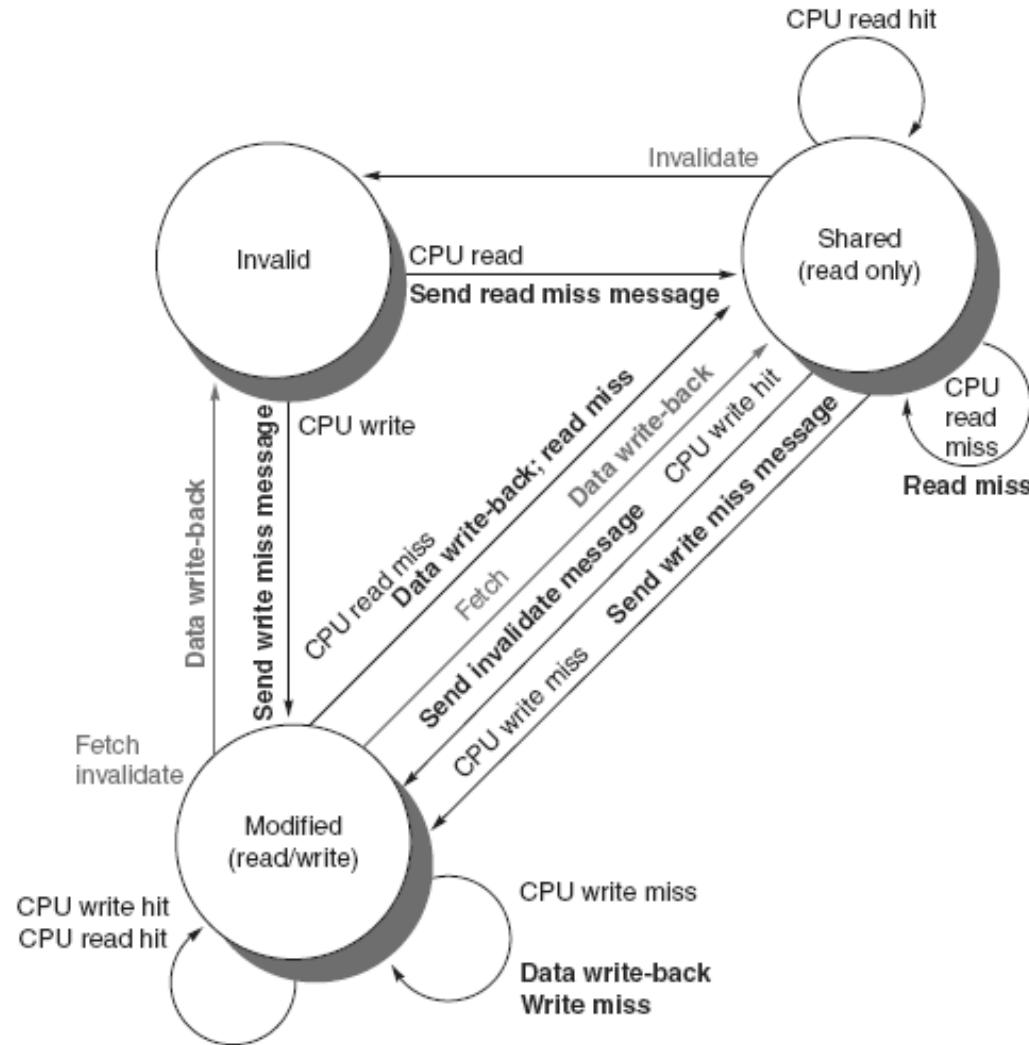
Directory Protocols

- For each block, maintain state:
 - Shared
 - One or more nodes have the block cached, value in memory is up-to-date
 - Set of node IDs
 - Uncached
 - Modified
 - Exactly one node has a copy of the cache block, value in memory is out-of-date
 - Owner node ID
- Directory maintains block states and sends invalidation messages

Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Directory Protocols



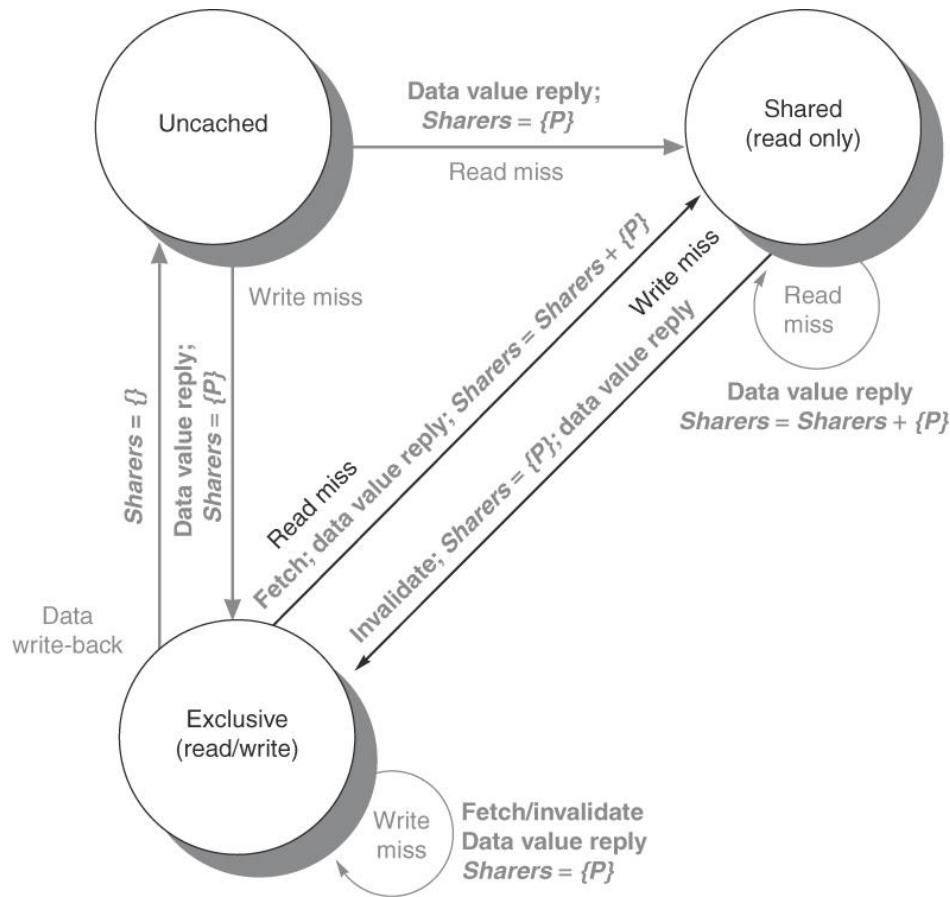


Figure 5.23 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.

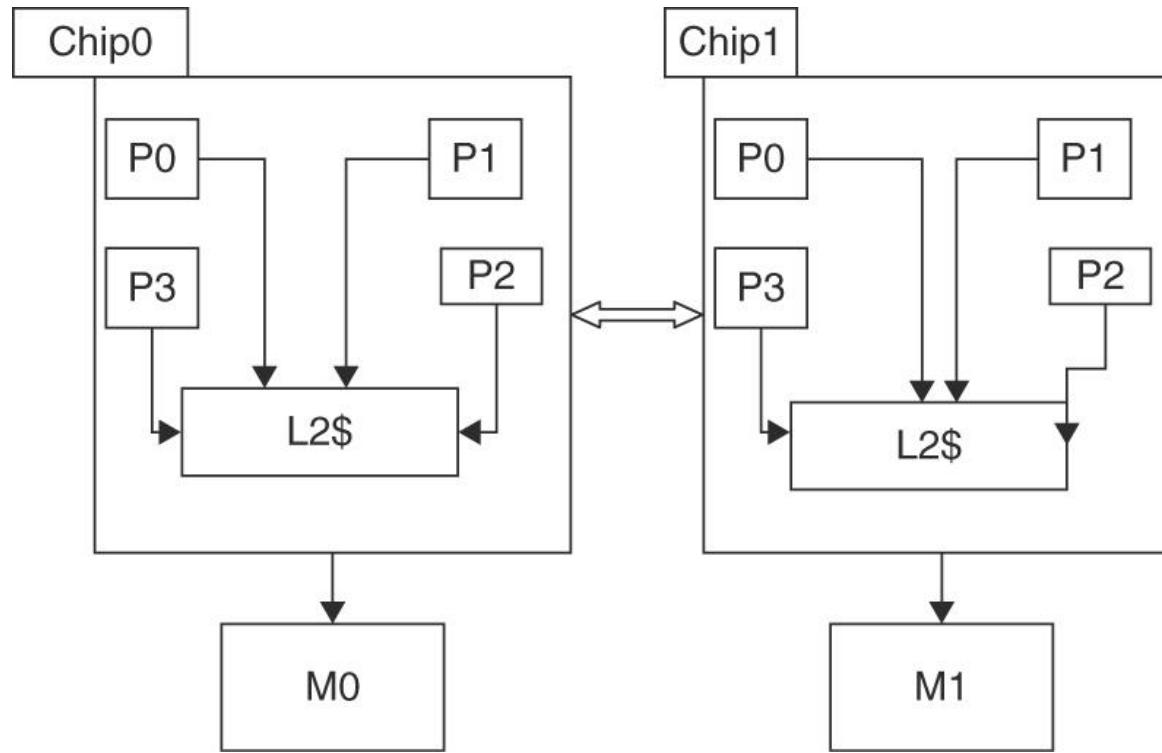
Directory Protocols

- For uncached block:
 - Read miss
 - Requesting node is sent the requested data and is made the only sharing node, block is now shared
 - Write miss
 - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
 - Read miss
 - The requesting node is sent the requested data from memory, node is added to sharing set
 - Write miss
 - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

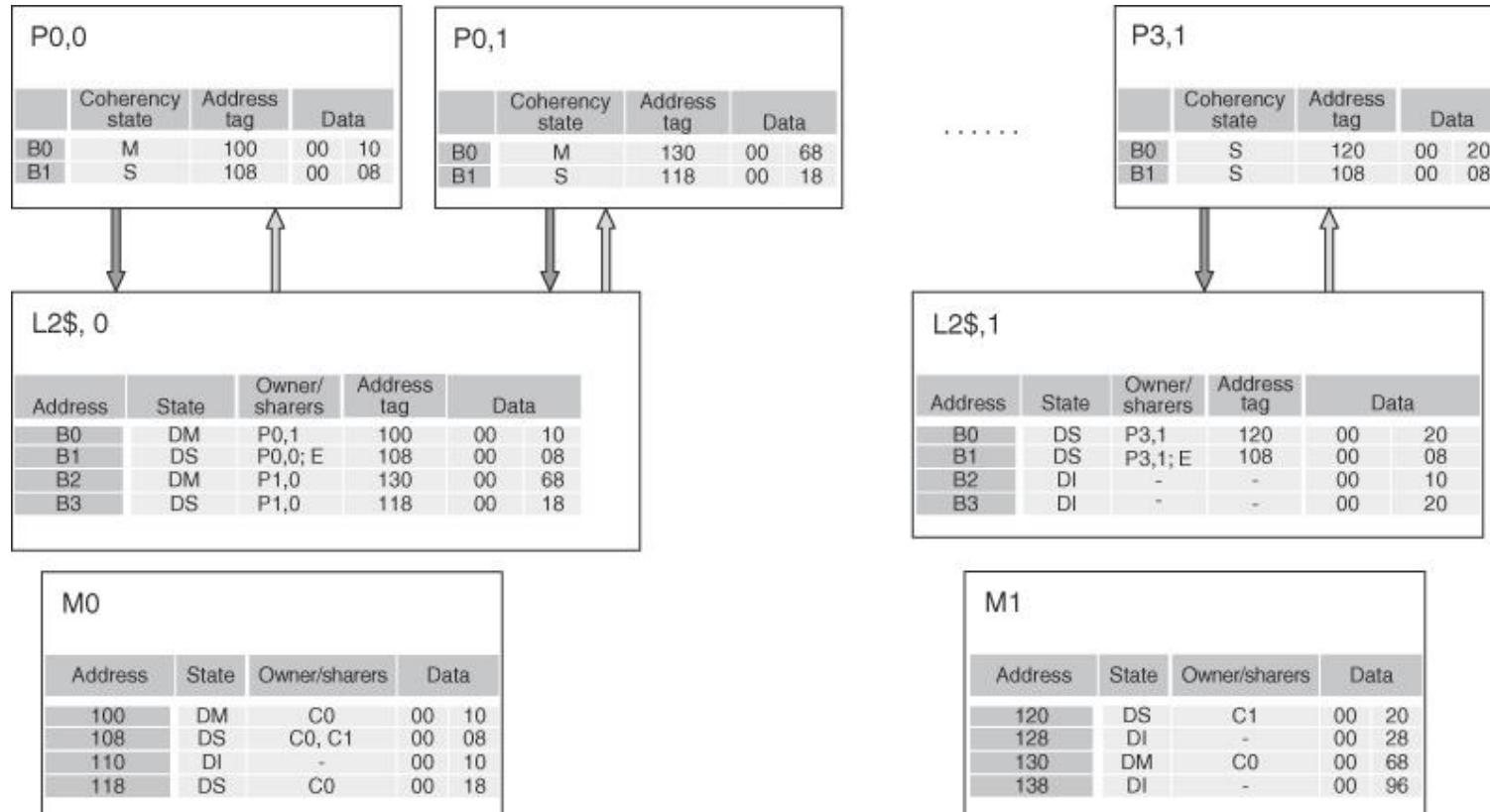
- For exclusive block:
 - Read miss
 - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
 - Data write back
 - Block becomes uncached, sharer set is empty
 - Write miss
 - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Example for Distributed Shared Memory Multiprocessors



Multichip, multicore multiprocessor with DSM.

Example for Distributed Shared Memory Multiprocessors



Cache and memory states in the multichip, multicore multiprocessor.

P0,0: read 100 L1 hit returns 0x0010, state unchanged (M)

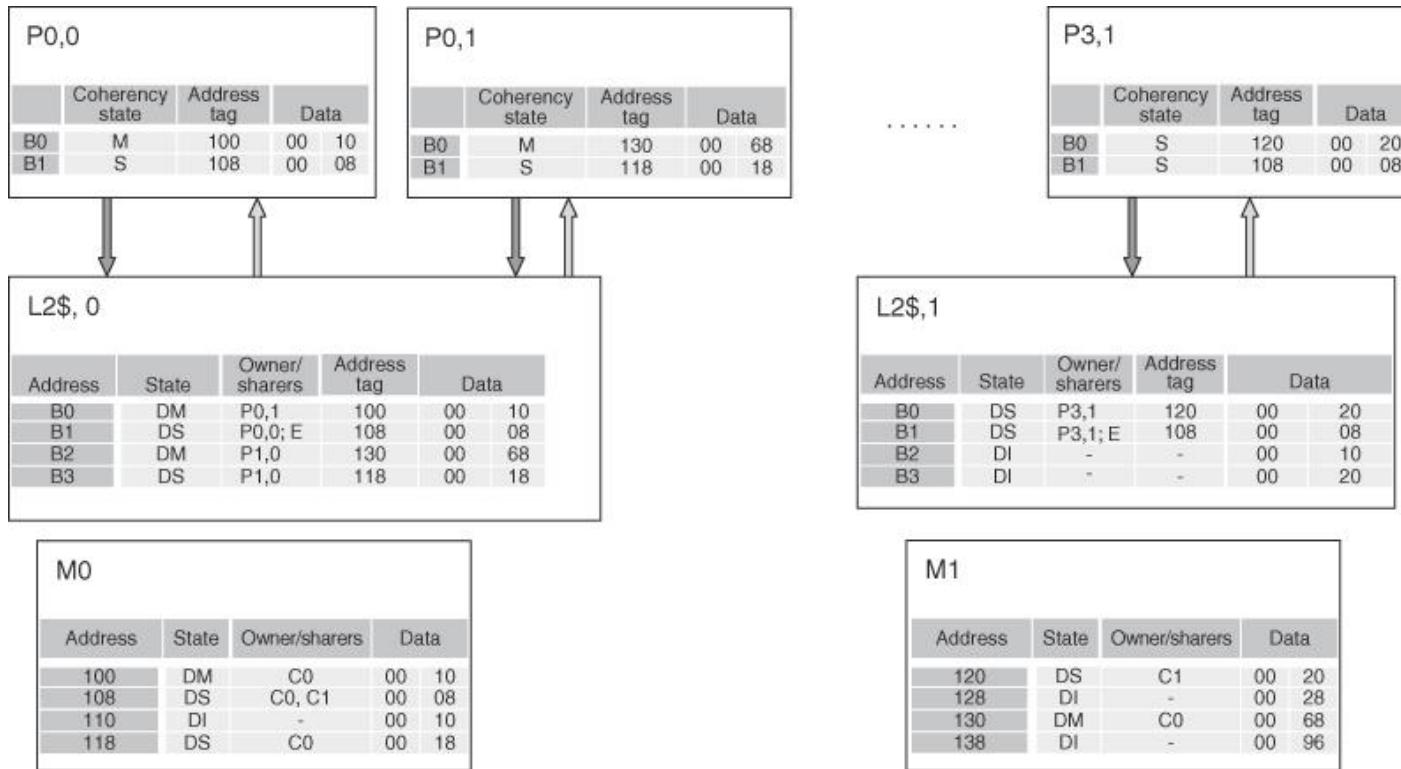
P0,0: read 128 L1 miss and L2 miss will replace B1 in L1 and B1 in L2 which has address 108.

L1 will have 128 in B1 (shared), L2 also will have it (DS, P0,0)

Memory directory entry for 108 will become <DS, C1>

Memory directory entry for 128 will become <DS, C0>

Example - DSM



P0,0: write 100 ← 80, Write hit only seen by P0,0

P0,0: write 108 ← 88, Write “upgrade” received by P0,0; invalidate received by P3,1

P0,0: write 118 ← 90, Write miss received by P0,0; invalidate received by P1,0

P1,0: write 128 ← 98, Write miss received by P1,0.

Synchronization

- Basic building blocks:
 - Atomic exchange
 - Swaps register with memory location
 - Test-and-set
 - Sets under condition
 - Fetch-and-increment
 - Reads original value from memory and increments it in memory
 - Requires memory read and write in uninterruptable instruction
- load linked/store conditional
 - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Implementing Spin Locks

- Spin lock (spinning around a loop until successful)
 - If no coherence: (lock variables are in memory)

	DADDUI	R2,R0,#1	
lockit:	EXCH	R2,0(R1)	;atomic exchange (write misses)
	BNEZ	R2,lockit	;already locked?

- If coherence: (maintain lock value coherently in the cache)
 - Read operations on the local copy of the lock until lock is available:

lockit:	LD	R2,0(R1)	;load of lock
	BNEZ	R2,lockit	;not available-spin
	DADDUI	R2,R0,#1	;load locked value
	EXCH	R2,0(R1)	;swap
	BNEZ	R2,lockit	;branch if lock wasn't 0

Implementing Locks

- Advantage of this scheme: reduces memory traffic

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0		None	

Models of Memory Consistency

- Memory consistency: In what order must a processor observe data writes of another processor?

<u>Processor 1:</u>	<u>Processor 2:</u>
A=0	B=0
...	...
A=1	B=1
if (B==0) ...	if (A==0) ...

- Should be impossible for both if-statements to be evaluated as true
 - Delayed write invalidate?
- Sequential consistency:
 - Result of execution should be the same as long as:
 - Accesses on each processor were kept in order
 - Accesses on different processors were arbitrarily interleaved

Implementing Locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
 - Reduces performance!
- Alternatives:
 - Program-enforced synchronization to force write on processor to occur before read on the other processor
 - Requires synchronization object for A and another for B
 - “Unlock” after write
 - “Lock” before read

Relaxed Consistency Models

- Allow reads/writes to complete out of order but use sync operations to enforce ordering and maintain sequential consistency
- Rules:
 - $X \rightarrow Y$
 - Operation X must complete before operation Y is done
 - Sequential consistency requires:
 - $R \rightarrow W, R \rightarrow R, W \rightarrow R, W \rightarrow W$
 - Relax $W \rightarrow R$
 - “Total store ordering”
 - Relax $W \rightarrow W$
 - “Partial store order”
 - Relax $R \rightarrow W$ and $R \rightarrow R$
 - “Weak ordering” and “release consistency”

Relaxed Consistency Models

- Consistency model is multiprocessor specific
- Programmers will often implement explicit synchronization
- Speculation gives much of the performance advantage of relaxed models with sequential consistency
 - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery

ECE 586

Hardware Security and Advanced Computer Architecture

LECTURE 21:

Data Level Parallelism in Vector and SIMD Architectures

04/24/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
 - Matrix-oriented scientific computing
 - Media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
 - Only needs to fetch one instruction per data operation
 - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- For x86 processors:
 - Expect two additional cores per chip per year
 - SIMD width to double every four years
 - Potential speedup from SIMD to be twice that from MIMD!

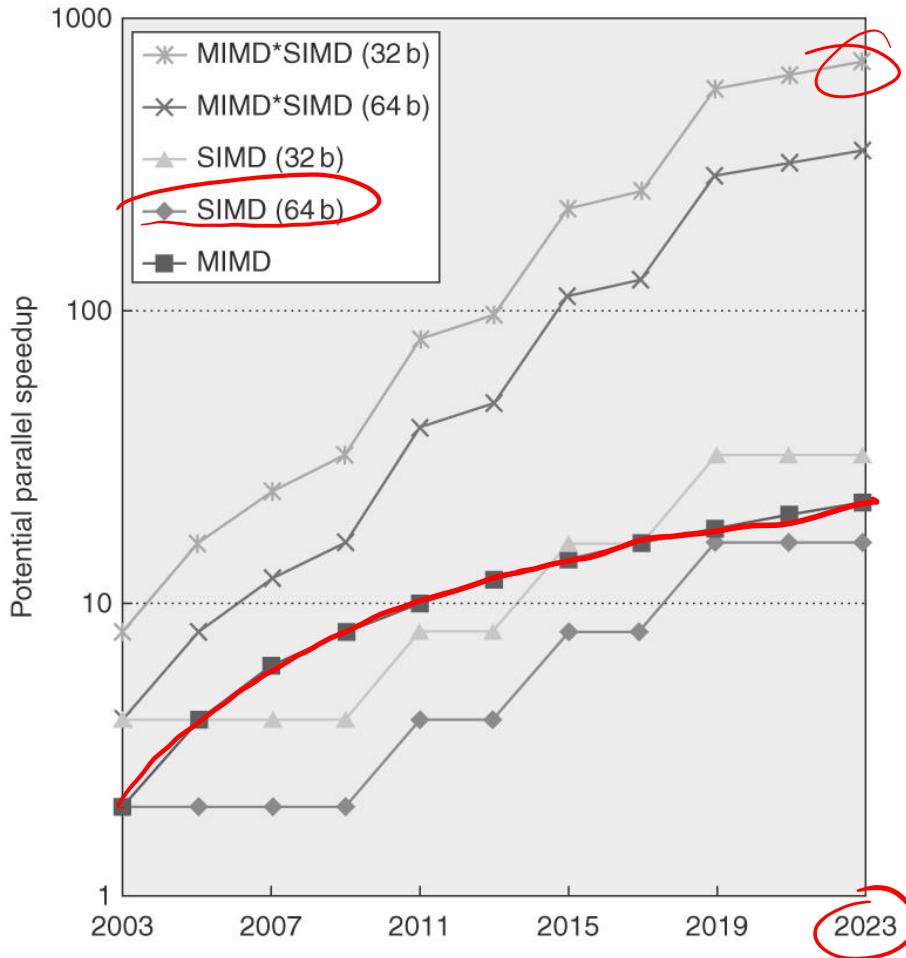


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

Vector Architectures

- Basic idea:
 - Read sets of data elements into "vector registers"
 - Operate on those registers
 - Disperse the results back into memory
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth

RV64V

- Example architecture: RV64V
 - Loosely based on Cray-1
 - 32 64-bit vector registers
 - Register file has 16 read ports and 8 write ports
 - Vector functional units
 - Fully pipelined
 - Data and control hazards are detected
 - Vector load-store unit
 - Fully pipelined
 - One word per clock cycle after initial latency
 - Scalar registers
 - 31 general-purpose registers
 - 32 floating-point registers

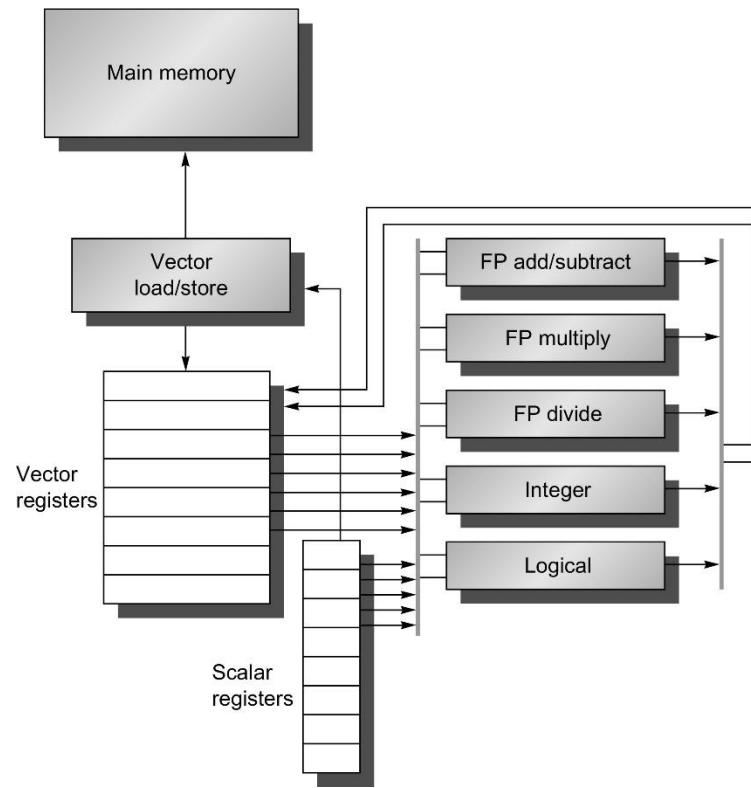


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. There are also 32 vector registers, and all the functional units are vector functional units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (*thick gray lines*) connects these ports to the inputs and outputs of the vector functional units.

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] from V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMAinder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare Root	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
yand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjk	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source register

Figure 4.2 The RV64V vector instructions. All use the R instruction format. Each vector operation with two operands is shown with both operands being vector (.vv), but there are also versions where the second operand is a scalar register (.vs) and, when it makes a difference, where the first operand is a scalar register and the second is a vector register (.sv). The type and width of the operands are determined by configuring each vector register rather than being supplied by the instruction. In addition to the vector registers and predicate registers, there are two vector control and status registers (CSRs), vl and vctype, discussed below. The strided and indexed data transfers are also explained later.

RV64V Instructions

- .vv: two vector operands
- .vs and .sv: vector and scalar operands
- vld/vst: vector load and vector store from address
- Example: DAXPY ($Y = a * X + Y$)

```
vsetdcfg    4*FP64      # Enable 4 DP FP vregs  
fld         f0,a        # Load scalar a  
vld         v0,x5       # Load vector X  
vmul        v1,v0,f0    # Vector-scalar mult  
vld         v2,x6       # Load vector Y  
vadd        v3,v1,v2    # Vector-vector add  
vst         v3,x6       # Store the sum  
vdisable
```

- 8 instructions, 258 for RV64V (scalar code)
 - Vector operations work on 32 elements

RISC-V DAXPY code

```
fld f0,a          # Load scalar a
addi x28,x5,#256 # Last address to load
Loop:  fld f1,0(x5)      # Load X[i]
       fmul.d f1,f1,f0    # a × X[i]
       fld f2,0(x6)      # Load Y[i]
       fadd.d f2,f2,f1    # a × X[i] + Y[i]
       fsd f2,0(x6)      # Store into Y[i]
       addi x5,x5,#8      # Increment index to X
       addi x6,x6,#8      # Increment index to Y
       bne x28,x5,Loop    # Check if done
```

Vector Execution Time

- Execution time depends on three factors:
 - Length of operand vectors
 - Structural hazards
 - Data dependencies
- RV64V functional units consume one element per clock cycle
 - Execution time is approximately the vector length
- *Convoy*
 - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards placed in same convoy via *chaining*
- *Chaining*
 - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
 - Unit of time to execute one convoy
 - m convoys executes in m chimes for vector length n
 - For vector length of n , requires $m \times n$ clock cycles

Example

```
vld          v0,x5      # Load vector X
vmul         v1,v0,f0   # Vector-scalar multiply
vld          v2,x6      # Load vector Y
vadd         v3,v1,v2   # Vector-vector add
vst          v3,x6      # Store the sum
```

Convoys:

1	vld	vmul
2	vld	vadd
3	vst	

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

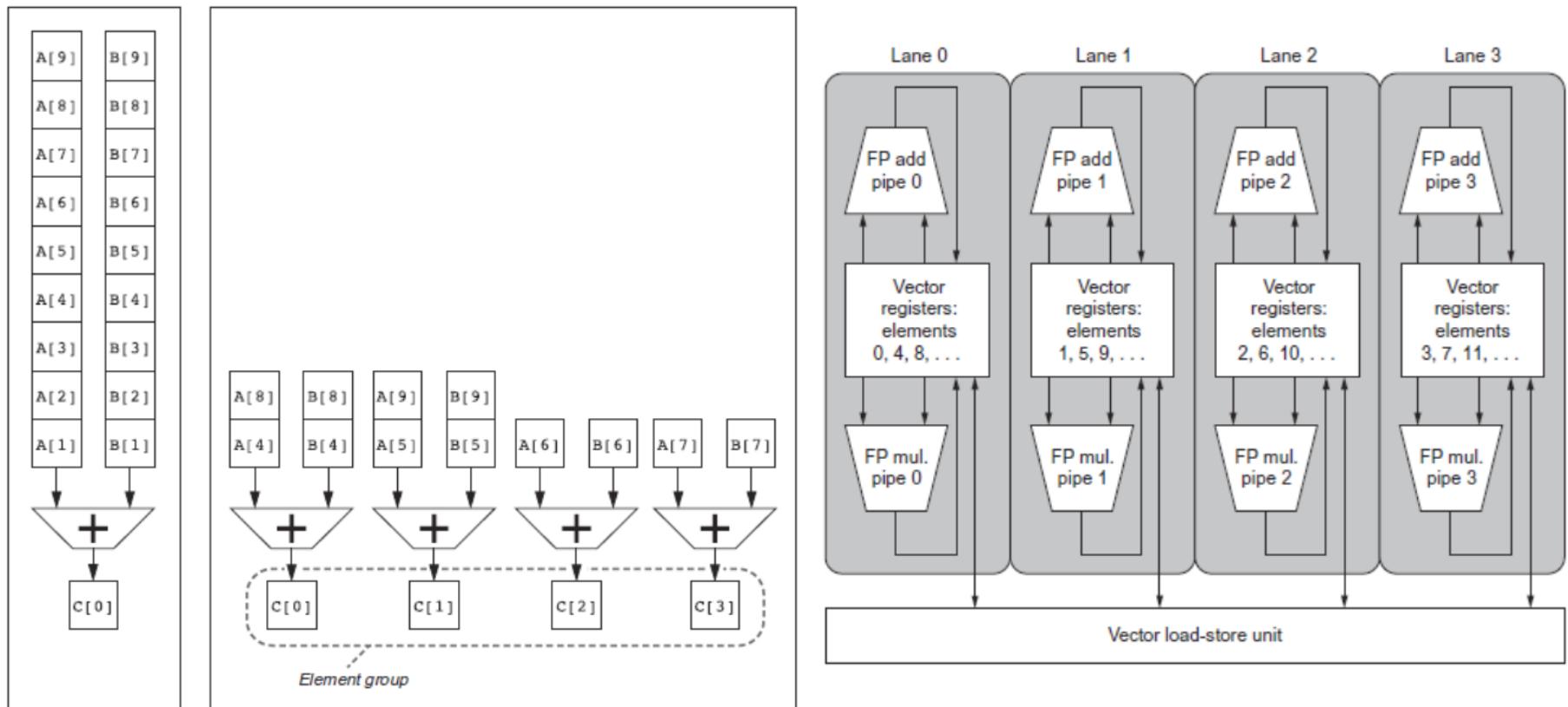
For 32 element vectors, requires $32 \times 3 = 96$ clock cycles

Challenges

- Start up time
 - Latency of vector functional unit
 - Assume the same as Cray-1
 - Floating-point add => 6 clock cycles
 - Floating-point multiply => 7 clock cycles
 - Floating-point divide => 20 clock cycles
 - Vector load => 12 clock cycles
- Improvements:
 - > 1 element per clock cycle
 - Non-64 wide vectors
 - IF statements in vector code
 - Memory system optimizations to support vector processors
 - Multiple dimensional matrices
 - Sparse matrices
 - Programming a vector computer

Multiple Lanes

- Element n of vector register A is “hardwired” to element n of vector register B
 - Allows for multiple hardware lanes



Vector Length Register and Strip Mining

```
for (i=0; i <n; i=i+1) Y[i] = a * X[i] + Y[i];
    vsetdcfg 2 DP FP          # Enable 2 64b Fl.Pt. registers
    fld f0,a                  # Load scalar a
loop:setvl t0,a0            # vl = t0 = min(mvl,n)
    vld v0,x5                # Load vector X
    slli t1,t0,3              # t1 = vl * 8 (in bytes)
    add x5,x5,t1             # Increment pointer to X by vl*8
    vmul v0,v0,f0             # Vector-scalar mult
    vld v1,x6                # Load vector Y
    vadd v1,v0,v1             # Vector-vector add
    sub a0,a0,t0              # n -= vl (t0)
    vst v1,x6                # Store the sum into Y
    add x6,x6,t1             # Increment pointer to Y by vl*8
    bnez a0,loop              # Repeat if n != 0
    vdisable                 # Disable vector regs}
```

Vector Mask Registers

- Consider:

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] - Y[i];
```

- Use predicate register to “disable” elements:

vsetdcfg	2*FP64	# Enable 2 64b FP vector regs
vsetpcfgi	1	# Enable 1 predicate register
vld	v0,x5	# Load vector X into v0
vld	v1,x6	# Load vector Y into v1
fmv.d.x	f0,x0	# Put (FP) zero into f0
vpne	p0,v0,f0	# Set p0(i) to 1 if v0(i)!=f0
vsub	v0,v0,v1	# Subtract under vector mask
vst	v0,x5	# Store the result in X
vdisable		# Disable vector registers
vpdisable		# Disable predicate registers

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
 - Control bank addresses independently
 - Load or store non sequential words (need independent bank addressing)
 - Support multiple vector processors sharing the same memory
- Example:
 - 32 processors, each generating 4 loads and 2 stores/cycle
 - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
 - How many memory banks needed?
 - $32 \times (4+2) \times 15 / 2.167 = \sim 1330$ banks

Stride

- Consider:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*
 - *vlds (strided load): Load from address R[rs1] with stride in R[rs2]:*
$$R[rs1] + i*R[rs2]$$
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
 - $\# \text{banks} / \text{LCM}(\text{stride}, \# \text{banks}) < \text{bank busy time}$

Scatter-Gather (sparse matrices)

- Consider:

```
for (i = 0; i < n; i=i+1)
```

```
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Use index vector:

vsetdcfg	4*FP64	# 4 64b FP vector registers
vld	v0, x7	# Load K[]
vldx	v1, x5, v0	# Load A[K[]]
vld	v2, x28	# Load M[]
vldi	v3, x6, v2	# Load C[M[]]
vadd	v1, v1, v3	# Add them
vstx	v1, x5, v0	# Store A[K[]]
vdisable		# Disable vector registers

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

SIMD Extensions

- Media applications operate on data types narrower than the native word size
 - Example: disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
 - Number of data operands encoded into op code
 - No sophisticated addressing modes (strided, scatter-gather)
 - No mask registers

SIMD Implementations

- Implementations:
 - Intel MMX (1996)
 - Eight 8-bit integer ops or four 16-bit integer ops
 - Streaming SIMD Extensions (SSE) (1999)
 - Eight 16-bit integer ops
 - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
 - Advanced Vector Extensions (2010)
 - Four 64-bit integer/fp ops
 - AVX-512 (2017)
 - Eight 64-bit integer/fp ops
 - Operands must be consecutive and aligned memory locations

Example SIMD Code

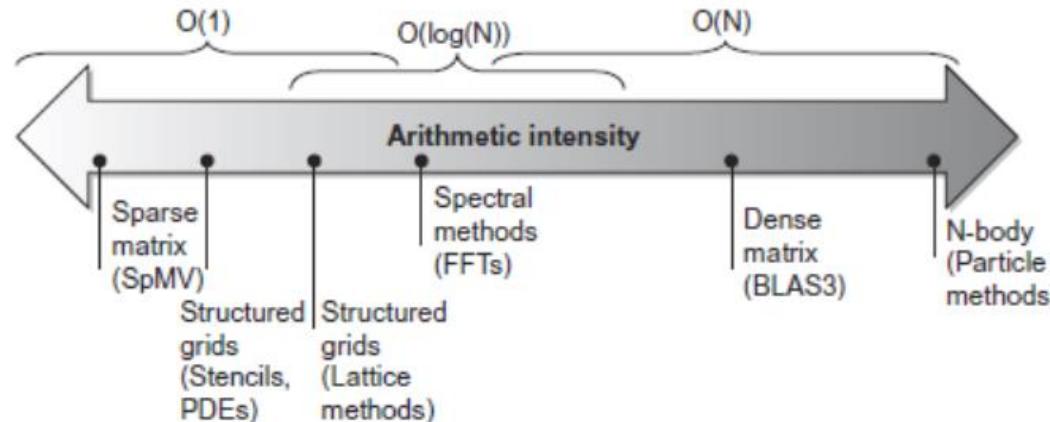
- Example DXPY:

```
fld      f0,a          # Load scalar a
splat.4D f0,f0         # Make 4 copies of a
addi    x28,x5,#256   # Last address to load
Loop: fld.4D  f1,0(x5)  # Load X[i] ... X[i+3]
       fmul.4D f1,f1,f0  # a x X[i] ... a x X[i+3]
       fld.4D   f2,0(x6)  # Load Y[i] ... Y[i+3]
       fadd.4D f2,f2,f1   # a x X[i]+Y[i]...
                           # a x X[i+3]+Y[i+3]
       fsd.4D   f2,0(x6)  # Store Y[i]... Y[i+3]
       addi    x5,x5,#32   # Increment index to X
       addi    x6,x6,#32   # Increment index to Y
       bne     x28,x5,Loop  # Check if done
```

67 instructions – 4x reduction compared to RISC-V

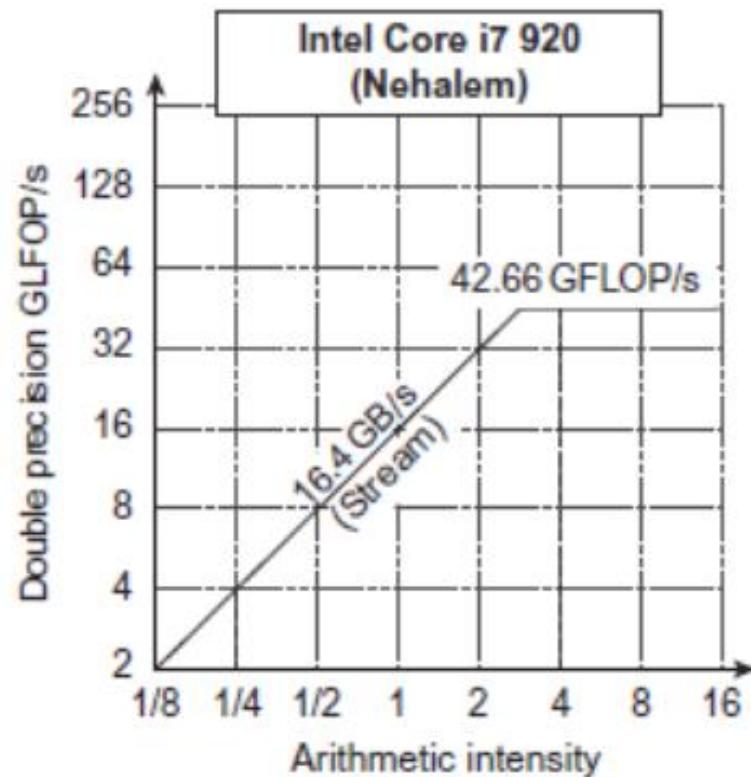
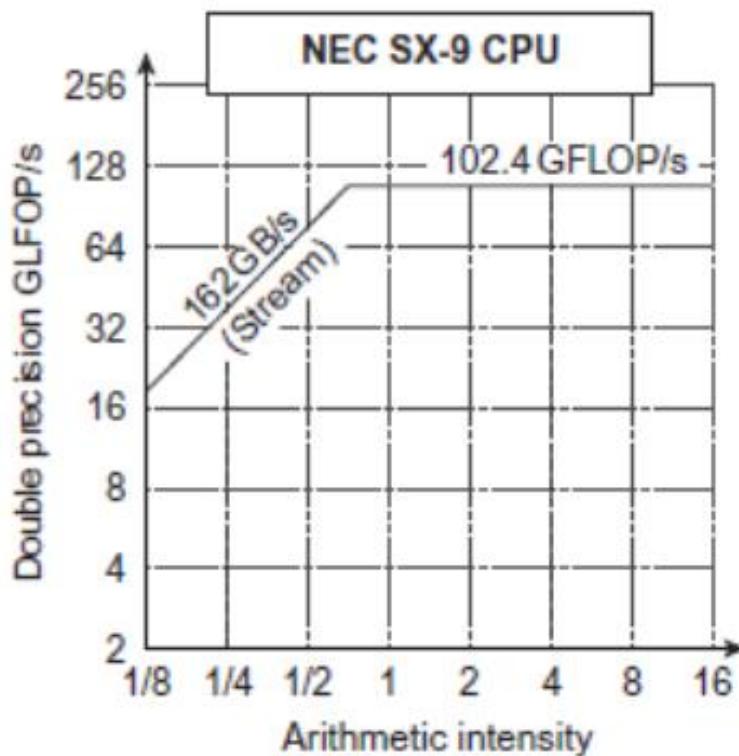
Roofline Performance Model

- Basic idea:
 - Plot peak floating-point throughput as a function of arithmetic intensity
 - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
 - Floating-point operations per byte read



Examples

- Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)



ECE 586 Hardware Security and Advanced Computer Architectures

LECTURE 22: **GPU Architectures**

04/26/2023

Erdal Oruklu, PhD

Illinois Institute of Technology
Department of Electrical and Computer Engineering

Graphical Processing Units

- Basic idea:
 - Heterogeneous execution model
 - CPU is the *host*, GPU is the *device*
 - Develop a C-like programming language for GPU
 - Unify all forms of GPU parallelism as *CUDA thread*
 - Programming model is “Single Instruction Multiple Thread”

Threads and Blocks

- A thread is associated with each data element
 - Threads are organized into blocks
 - Blocks are organized into a grid
-
- GPU hardware handles thread management, not applications or OS

NVIDIA GPU Architecture

- Similarities to vector machines:
 - Works well with data-level parallel problems
 - Scatter-gather transfers
 - Mask registers
 - Large register files
- Differences:
 - No scalar processor
 - Uses multithreading to hide memory latency
 - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

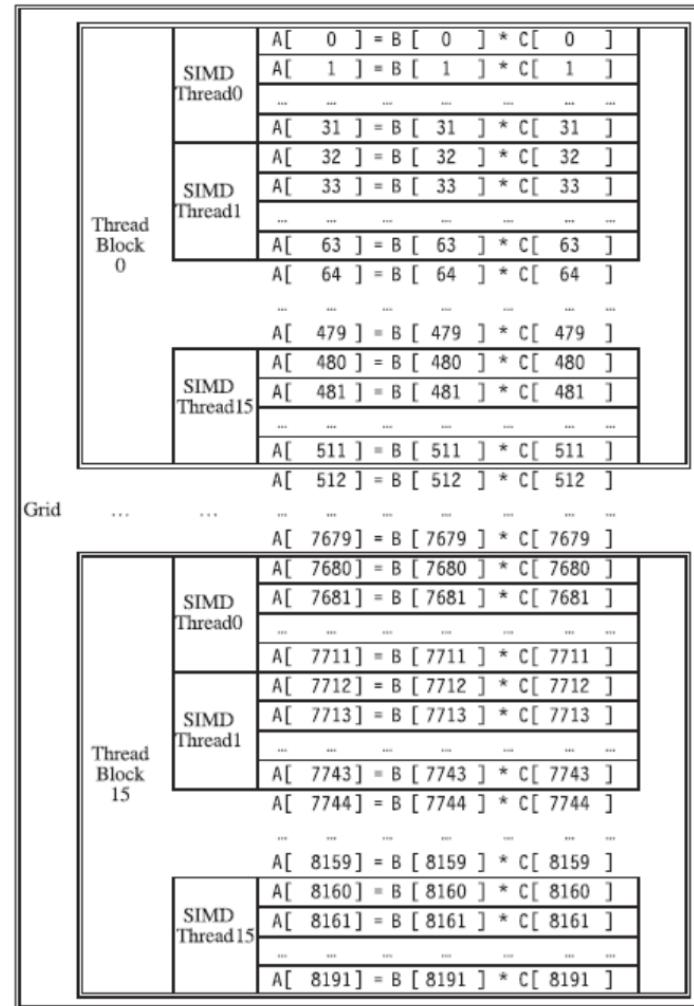
Example (Pascal)

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
 - 512 elements per block
- SIMD instruction executes 32 elements at a time
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Current-generation GPUs have 84 multithreaded SIMD processors

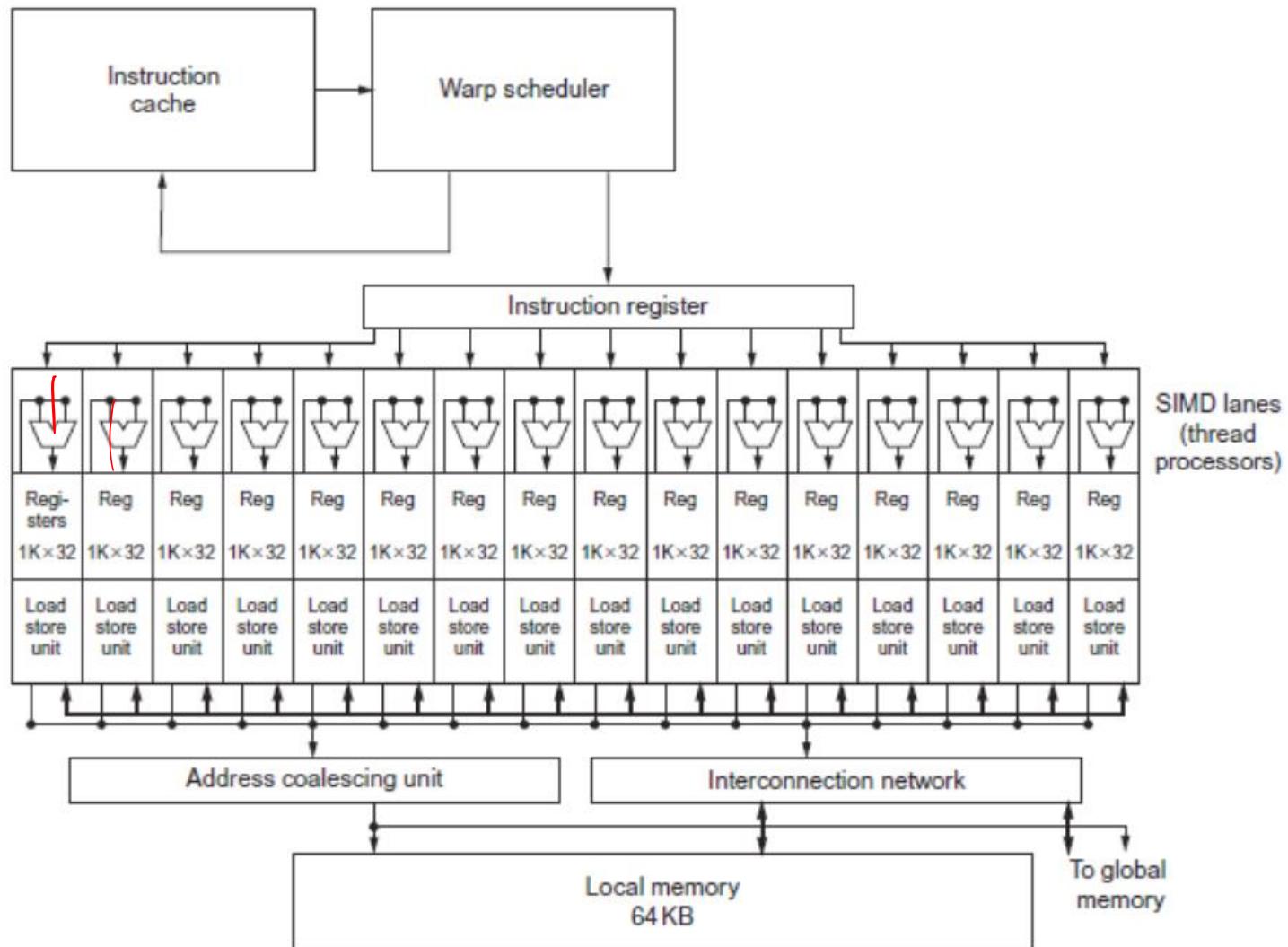
Terminology

- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or “warp”
 - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
 - Each warp has its own PC
 - Thread scheduler uses scoreboard to dispatch warps
 - By definition, no data dependencies between warps
 - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
 - 32 SIMD lanes
 - Wide and shallow compared to vector processors

Example



GPU Organization



NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
 - “Parallel Thread Execution (PTX) ”
 - opcode.type d,a,b,c;
 - Uses virtual registers
 - Translation to machine code is performed in software
 - Example:

```
shl.s32      R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.s32      R8, R8, threadIdx   ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]       ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]       ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4          ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2          ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0       ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
 - Branch synchronization stack
 - Entries consist of masks for each SIMD lane
 - I.e. which threads commit their results (all threads execute)
 - Instruction markers to manage when a branch diverges into multiple execution paths
 - Push on divergent branch
 - ...and when paths converge
 - Act as barriers
 - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

```
if (X[i] != 0)
    X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

ld.global.f64	RD0, [X+R8]	; RD0 = X[i]
setp.neq.s32	P1, RD0, #0	; P1 is predicate register 1
@!P1, bra	ELSE1, *Push	; Push old mask, set new mask bits
ld.global.f64	RD2, [Y+R8]	; RD2 = Y[i]
sub.f64	RD0, RD0, RD2	; Difference in RD0
st.global.f64	[X+R8], RD0	; X[i] = RD0
@P1, bra	ENDIF1, *Comp	; complement mask bits
ELSE1:	ld.global.f64 RD0, [Z+R8]	; RD0 = Z[i]
	st.global.f64 [X+R8], RD0	; X[i] = RD0
ENDIF1:	<next instruction>, *Pop	; pop to restore old mask

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
 - “Private memory”
 - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
 - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
 - Host can read and write GPU memory

NVIDIA GPU Memory Structures

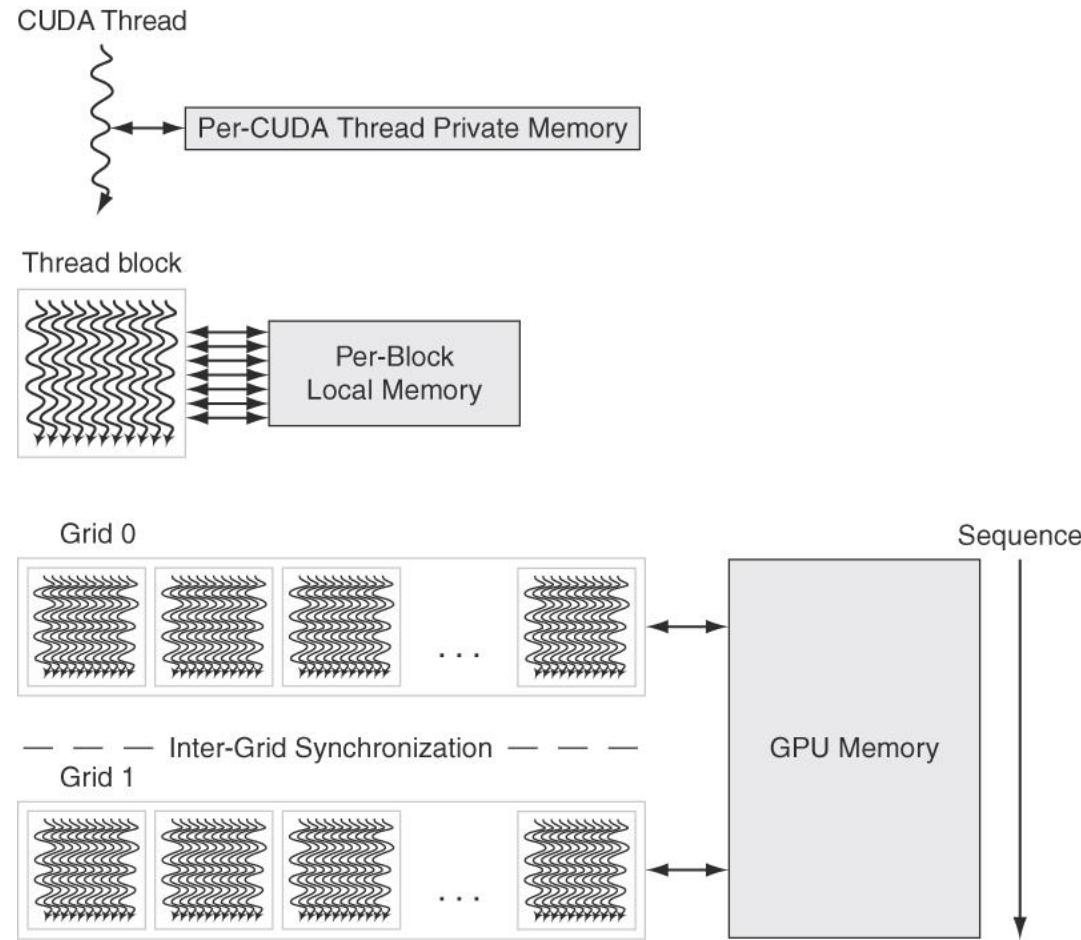


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Pascal Architecture Innovations

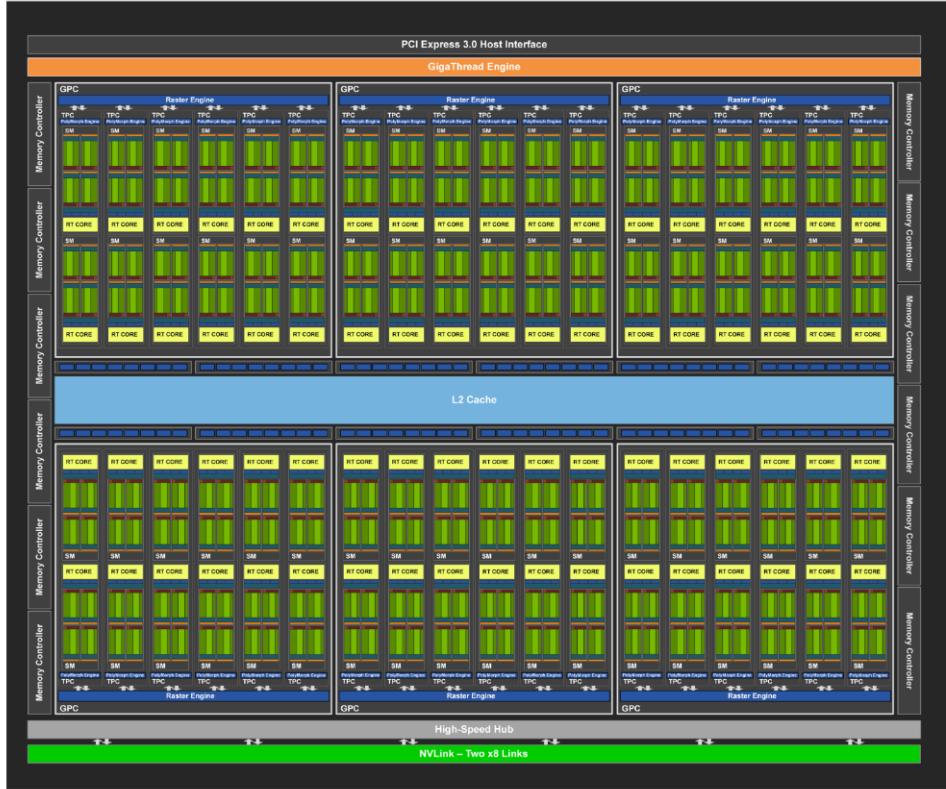
- Each SIMD processor has
 - Two or four SIMD thread schedulers, two instruction dispatch units
 - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
 - Two threads of SIMD instructions are scheduled every two clock cycles
- Fast single-, double-, and half-precision
- High Bandwidth Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

Pascal Multithreaded SIMD Proc.



Turing Architecture

- The TU102 GPU includes six Graphics Processing Clusters (GPCs), 36 Texture Processing Clusters (TPCs), and 72 Streaming Multiprocessors (SMs).
- The full implementation of the TU102 GPU includes the following:
 - 4,608 CUDA Cores
 - 72 RT Cores
 - 576 Tensor Cores
- Turing Tensor Cores accelerate the matrix-matrix multiplication at the heart of neural network training and inferencing functions.
 - Each Tensor Core can perform up to 64 floating point fused multiply-add (FMA) operations per clock using FP16 inputs. Eight Tensor Cores in an SM perform a total of 512 FP16 multiply and accumulate operations per clock, or 1024 total FP operations per clock. The INT8 precision mode works at double this rate, or 2048 integer operations per clock.



Turing SM

- The Turing SM is partitioned into four processing blocks, each with 16 FP32 Cores, 16 INT32 Cores, two Tensor Cores, one warp scheduler, and one dispatch unit.
- Each block includes a new L0 instruction cache and a 64 KB register file. The four processing blocks share a combined 96 KB L1 data cache/shared memory.



Ampere Architecture

- The full GA102 GPU contains seven GPCs, 42 TPCs, and 84 SMs.
- The full implementation of the GA102 GPU includes the following:
 - 10752 CUDA Cores
 - 84 RT Cores
 - 336 Tensor Cores



Ampere SM

- GA10x SM is partitioned into four processing blocks (or partitions), each with a 64 KB register file, an L0 instruction cache, one warp scheduler, one dispatch unit, and sets of math and other units. The four partitions share a combined 128 KB L1 data cache/shared memory subsystem.
- Unlike the TU102 SM which includes two second-generation Tensor Cores per partition and eight Tensor Cores total, the new GA10x SM includes one third-generation Tensor Core per partition and four Tensor Cores total, with each GA10x Tensor Core being twice as powerful as a Turing Tensor Core.
- GA10X includes FP32 processing on two datapaths, doubling the peak processing rate for FP32 operations.



Vector Architectures vs GPUs

- SIMD processor analogous to vector processor, both have MIMD
- Registers
 - RV64V register file holds entire vectors
 - GPU distributes vectors across the registers of SIMD lanes
 - RV64 has 32 vector registers of 32 elements (1024)
 - GPU has 256 registers with 32 elements each (8K)
 - RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles
 - SIMD processor chime is 2 to 4 cycles
 - GPU vectorized loop is grid
 - All GPU loads are gather instructions and all GPU stores are scatter instructions

Vector Processor vs GPUs

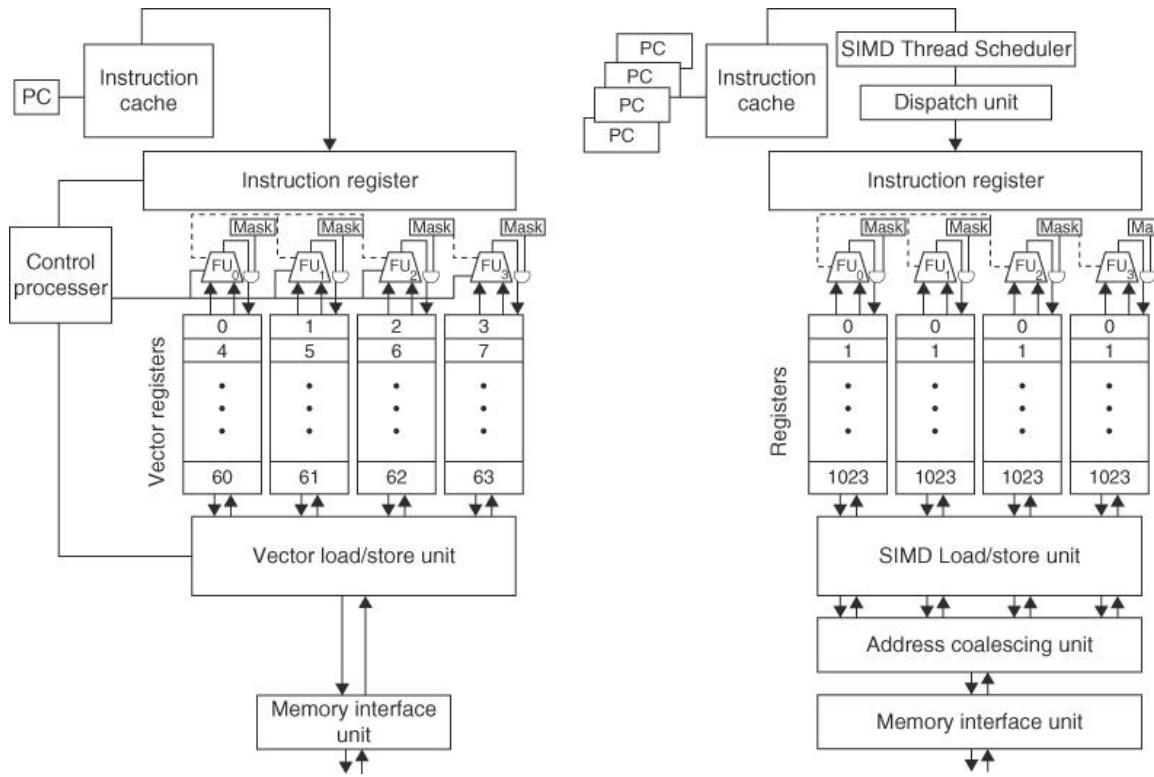


Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 8 to 16 SIMD Lanes.) The control processor supplies scalar operands for scalar-vector operations, increments addressing for unit and non-unit stride accesses to memory, and performs other accounting-type operations. Peak memory performance only occurs in a GPU when the Address Coalescing unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD thread to help with multithreading.

SIMD Architectures vs GPUs

- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

Fallacies and Pitfalls

- GPUs suffer from being coprocessors
 - GPUs have flexibility to change ISA
- Concentrating on peak performance in vector architectures and ignoring start-up overhead
 - Overheads require long vector lengths to achieve speedup
- Increasing vector performance without comparable increases in scalar performance
- You can get good vector performance without providing memory bandwidth
- On GPUs, just add more threads if you don't have enough memory performance

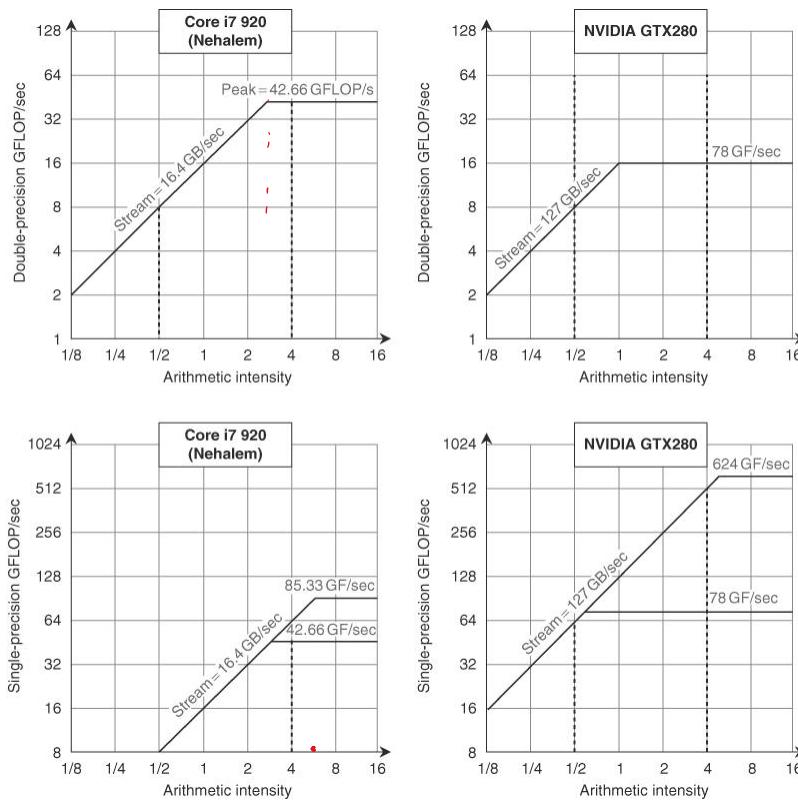


Figure 4.28 Roofline model [Williams et al. 2009]. These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/sec, a SP FP peak of 85.33 GFLOP/sec, and a peak memory bandwidth of 16.4 GBytes/sec. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/sec, SP FP peak of 624 GFLOP/sec, and 127 GBytes/sec of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/sec or 8 SP GFLOP/sec on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 42.66 DP GFLOP/sec and 64 SP GFLOP/sec on the Core i7 and 78 DP GFLOP/sec and 512 DP GFLOP/sec on the GTX 280. To hit the highest computation rate on the Core i7 you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors. Guz et al. [2009] have an interesting analytic model for these two architectures.