CSE 141: Introduction to Computer Architecture

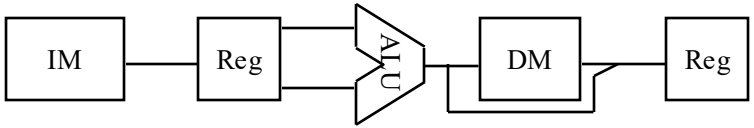
Memory & Caches

HOW DOES “MEMORY” WORK?

Many slides in this subsection borrowed from Prabal

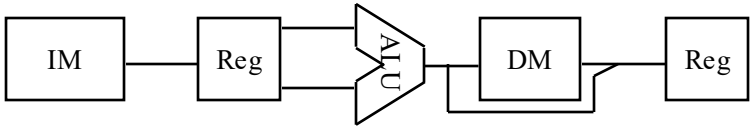
How does memory actually connect to the processor?

- We have been working with this view of the pipeline for a while:

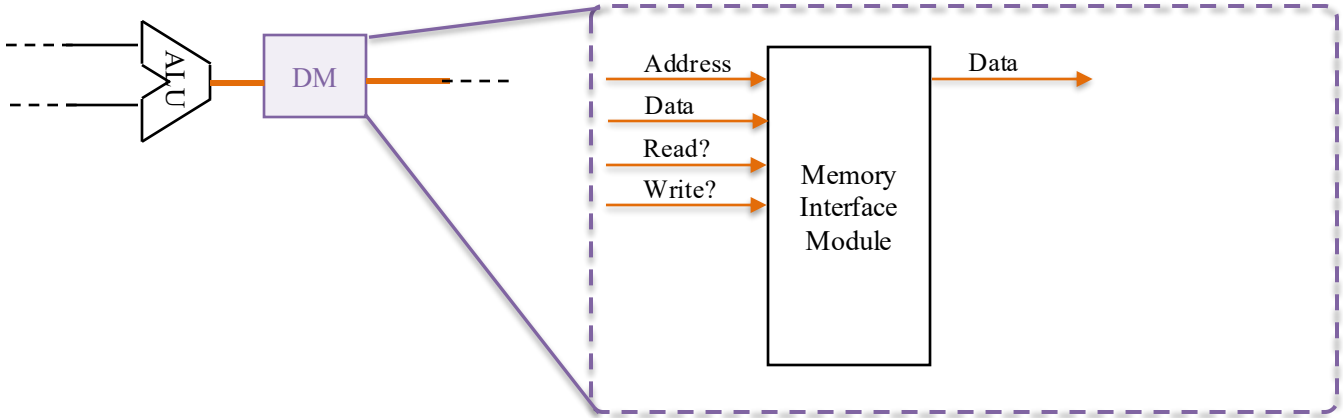


How does memory actually connect to the processor?

- We have been working with this view of the pipeline for a while:

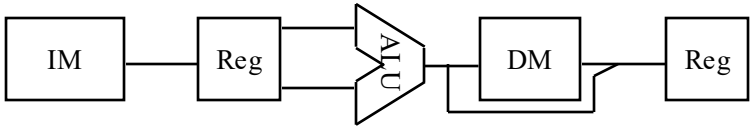


- For machines with no caches, the “DM” looks more like this:

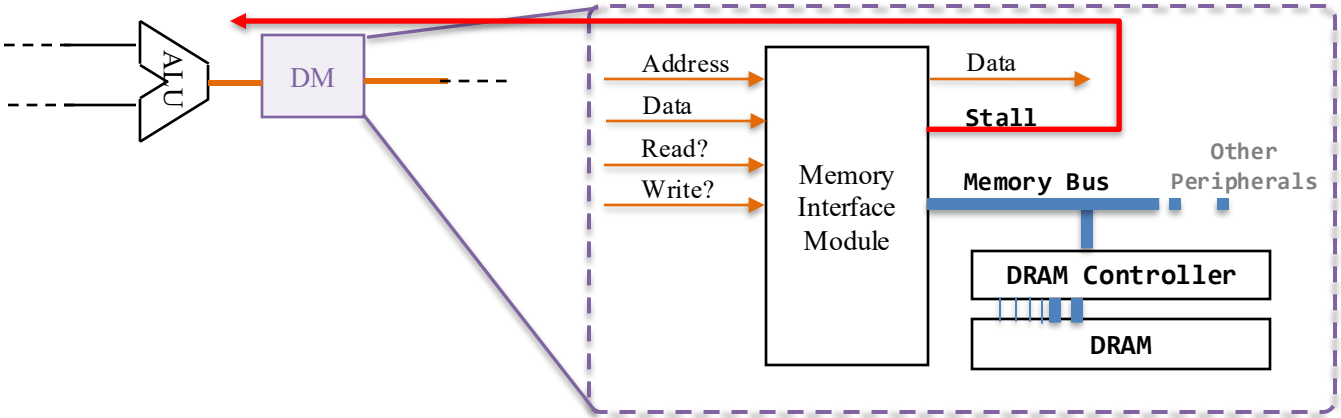


How does memory actually connect to the processor?

- We have been working with this view of the pipeline for a while:

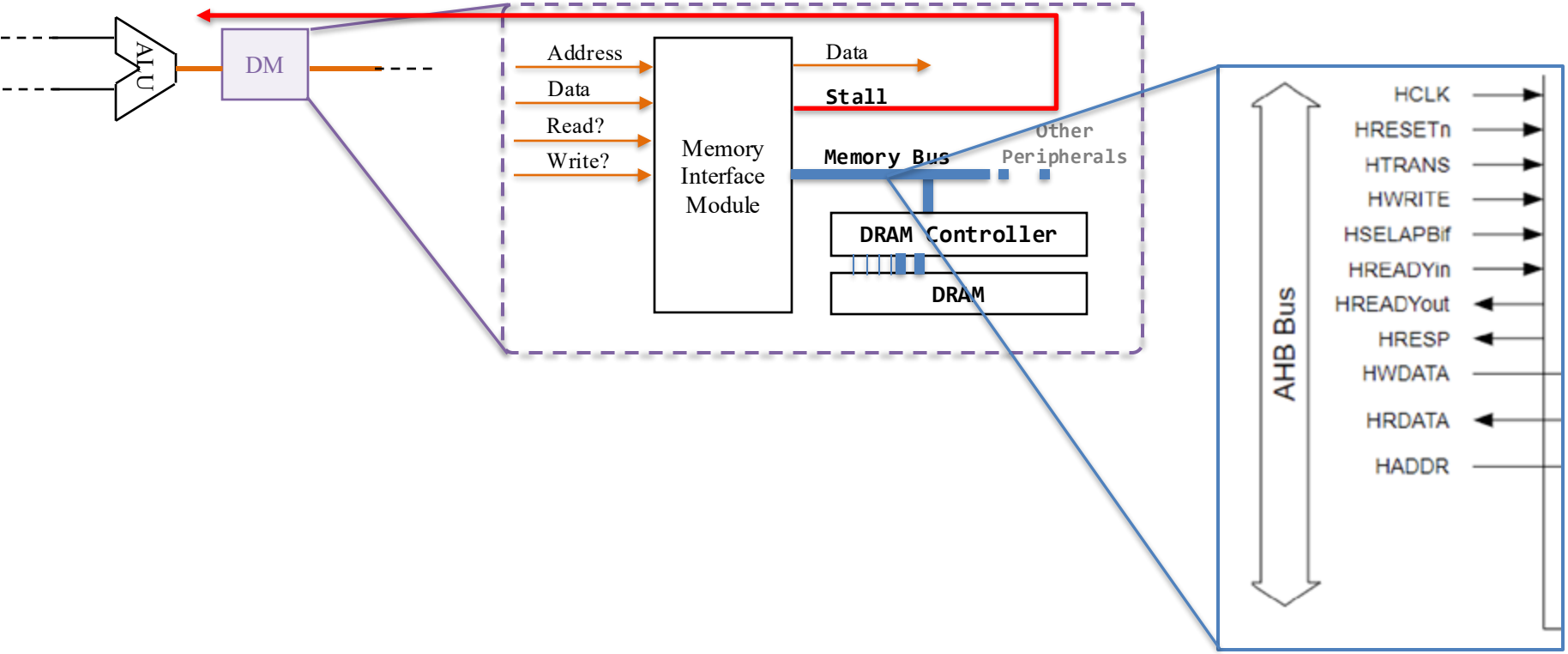


- For machines with no caches, the “DM” looks more like this:

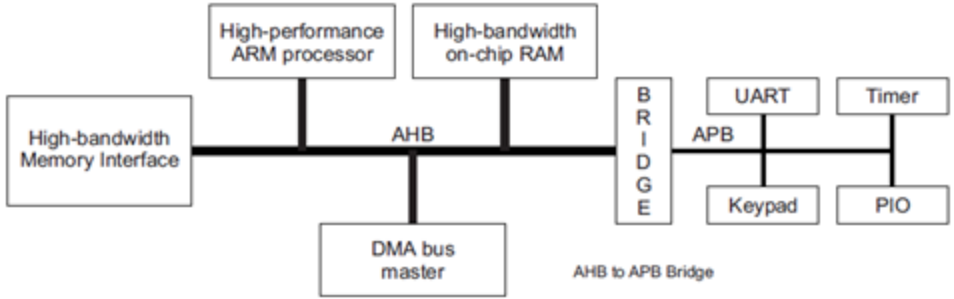


How does memory actually connect to the processor?

On ARM microcontrollers



Advanced Microcontroller Bus Architecture (AMBA)
- Advanced High-performance Bus (AHB)
- Advanced Peripheral Bus (APB)



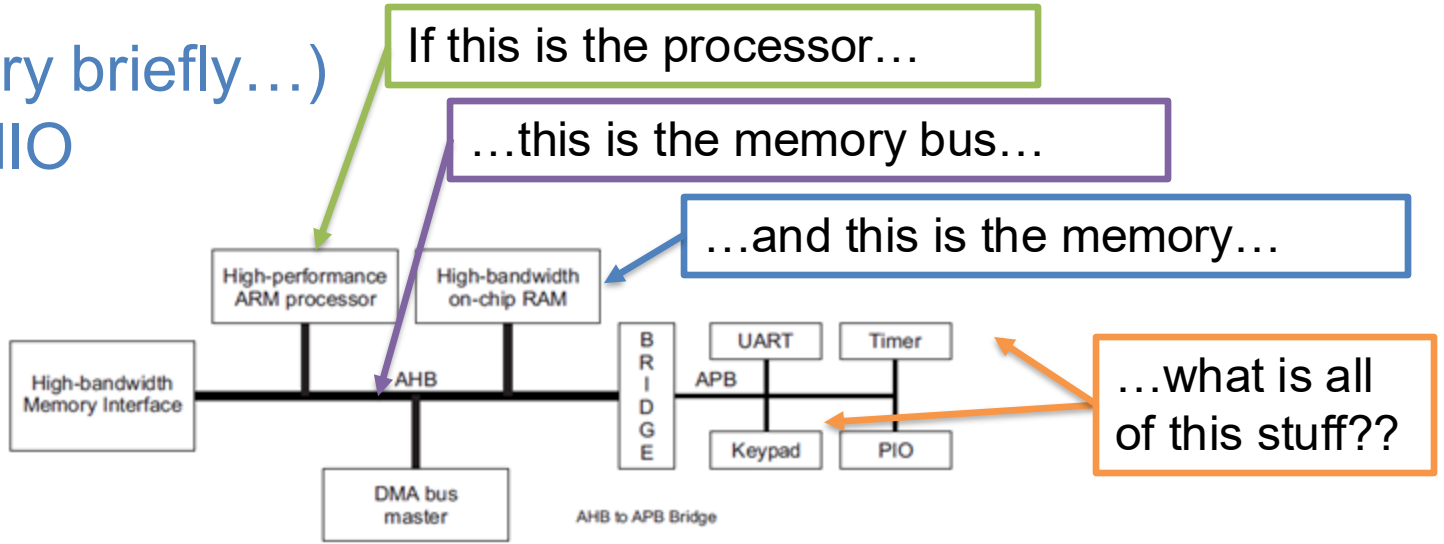
AHB

- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

APB

- Low power
- Latched address/control
- Simple interface
- Suitable of many peripherals

(Very briefly…) MMIO



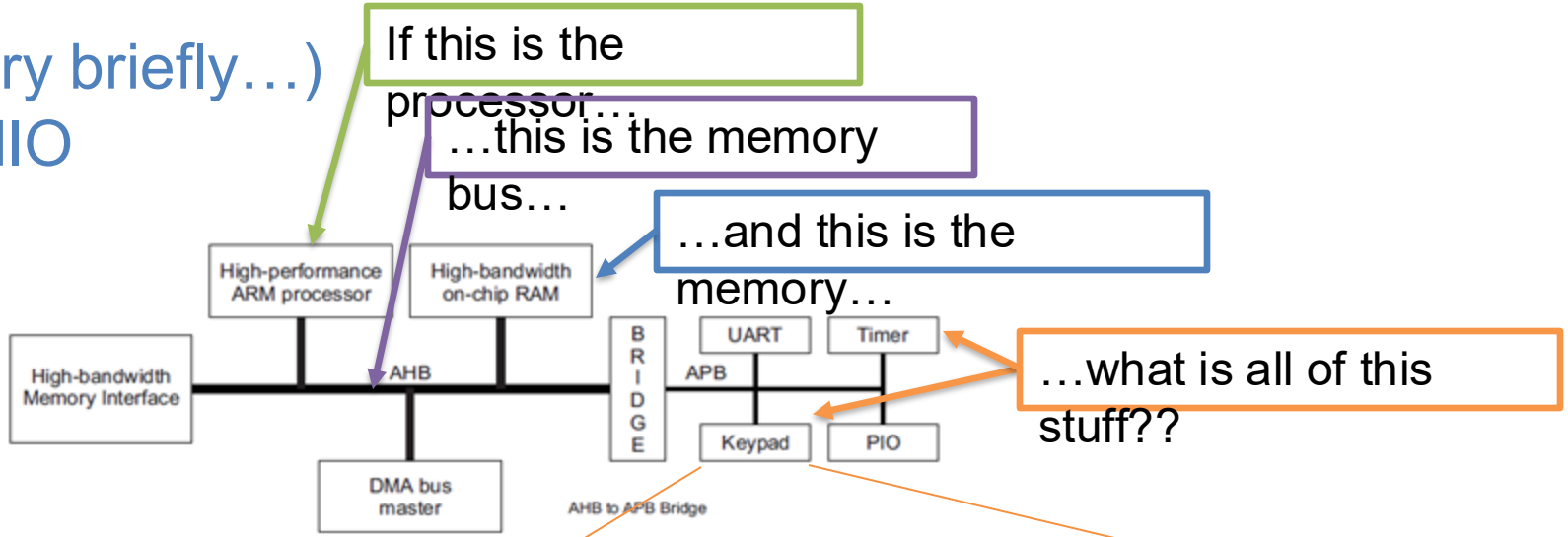
AHB

- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

APB

- Low power
- Latched address/control
- Simple interface
- Suitable of many peripherals

(Very briefly…) MMIO



"Memory-Mapped I/O"

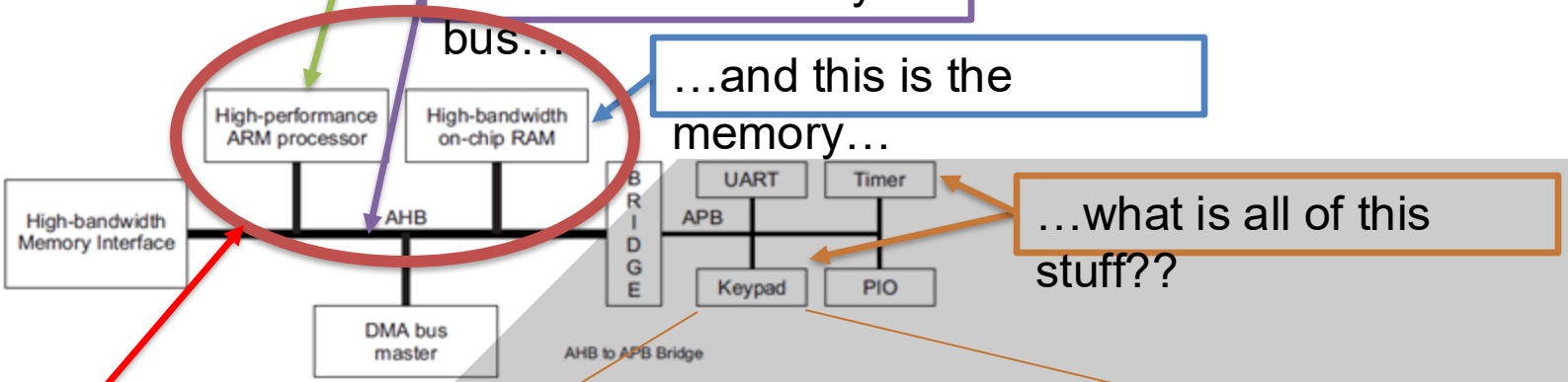
```
// Set the brightness of the keyboard backlight to 100%
*( (uint8_t*) 0x40001000) = 255;
// Read which key is currently pressed
uint8_t key = *( (uint8_t*) 0x4000101C);
```

For 141…

If this is the processor…

…this is the memory bus…

…and this is the memory…



…what is all of this stuff??

We're just going to look at examples of CPU<>RAM in this class, but the ideas hold for everything

"Memory-Mapped I/O"

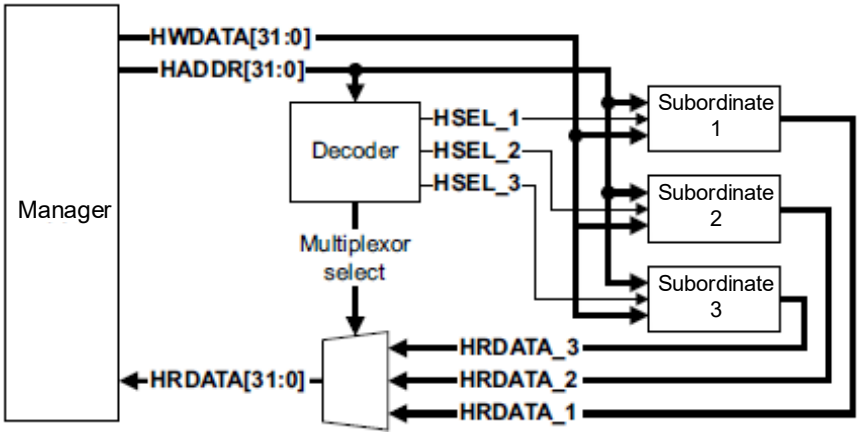
```
// Set the brightness of the keyboard backlight to 100%
*( (uint8_t*) 0x40001000) = 255;
// Read which key is currently pressed
uint8_t key = *( (uint8_t*) 0x4000101C);
```

A brief aside: Historical Terminology

- Many protocols used “master/slave” terms for a long time
- Most have renamed things in the last few years
  - ARM AMBA:
    - AHB: Master/Slave → Manager/Subordinate
    - APB: Master/Slave → Requester/Completer
    - … full list of ABMA terms: https://developer.arm.com/documentation/109550/latest/
  - Other examples:
    - SPI: Master/Slave → Controller/Peripheral
    - Bluetooth: Master/Slave → Central/Peripheral
    - MIPI/I3C/I2C: Master/Slave → Controller/Target or Leader/Follower
    - …
- In practice, the new names are often more descriptive and more useful
- Often only new documentation is updated, so you may see a mix of terms
  - (including in these slides; especially figures, where my ppt textbox editing is imperfect)

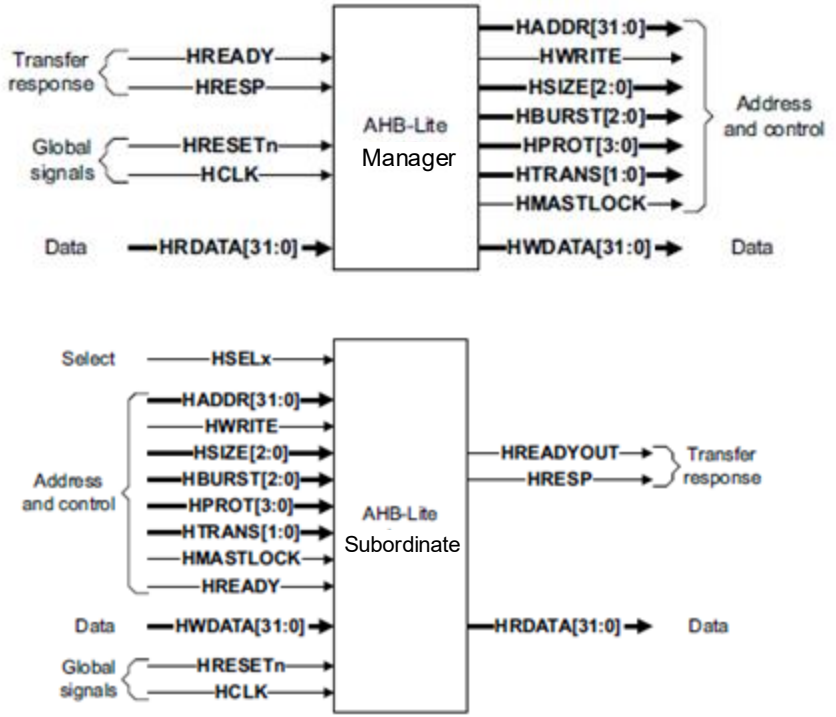
AHB-Lite supports single bus manager and provides high-bandwidth operation

- Burst transfers
- Single clock-edge operation
- Non-tri-state implementation
- Configurable bus width



AHB-Lite bus manager/subordinate interface

- Global signals
  - HCLK
  - HRESETn
- Manager out / subordinate in
  - HADDR (address)
  - HWDATA (write data)
  - Control
    - HWRITE
    - HSIZE
    - HBURST
    - HPROT
    - HTRANS
    - HMASTLOCK
- Subordinate out / manager in
  - HRDATA (read data)
  - HREADY
  - HRESP



Many slides in this subsection borrowed from Prabal

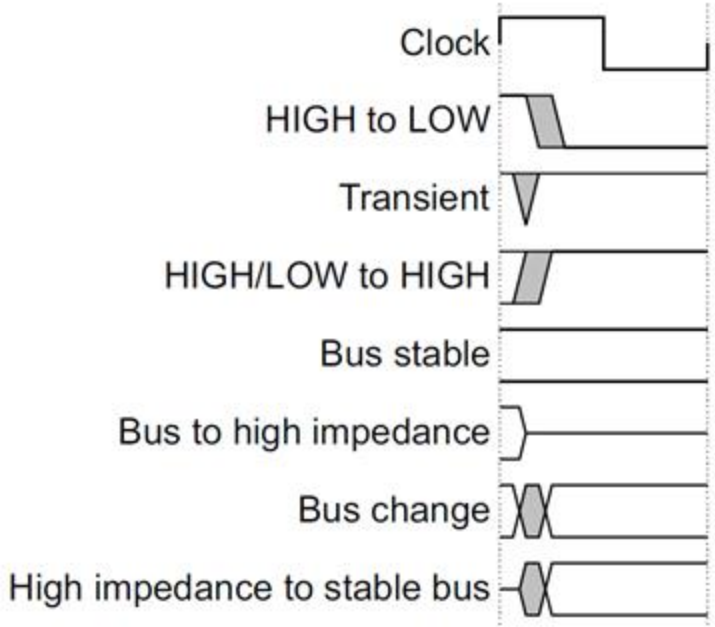
AHB-Lite signal definitions

- Global signals
  - HCLK: the bus clock source (rising-edge triggered)
  - HRESETn: the bus (and system) reset signal (active low)
- Manager out / Subordinate in
  - HADDR[31:0]: the 32-bit system address bus
  - HWDATA[31:0]: the system write data bus
  - Control
    - HWRITE: indicates transfer direction (Write=1, Read=0)
    - HSIZE[2:0]: indicates size of transfer (byte, halfword, or word)
    - HBURST[2:0]: indicates single or burst transfer (1, 4, 8, 16 beats)
    - HPROT[3:0]: provides protection information (e.g. I or D; user or handler)
    - HTRANS: indicates current transfer type (e.g. idle, busy, nonseq, seq)
    - HMASTLOCK: indicates a locked (atomic) transfer sequence
- Subordinate out / Manager in
  - HRDATA[31:0]: the subordinate read data bus
  - HREADY: indicates previous transfer is complete
  - HRESP: the transfer response (OKAY=0, ERROR=1)

Key to timing diagram conventions

- Timing diagrams
  - Clock
  - Stable values
  - Transitions
  - High-impedance

- Signal conventions
  - Lower case ‘n’ denote active low (e.g. RESETn)
  - Prefix ‘H’ denotes AHB
  - Prefix ‘P’ denotes APB



Basic read and write transfers with no wait states

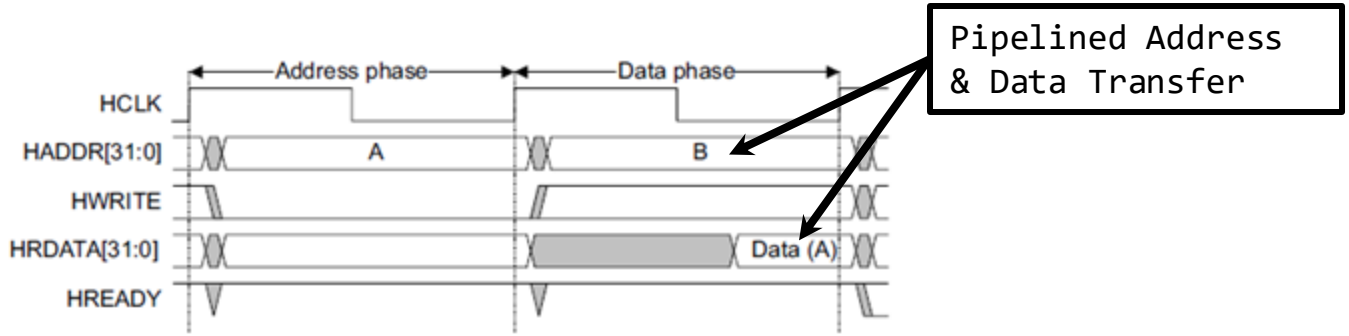


Figure 3-1 Read transfer

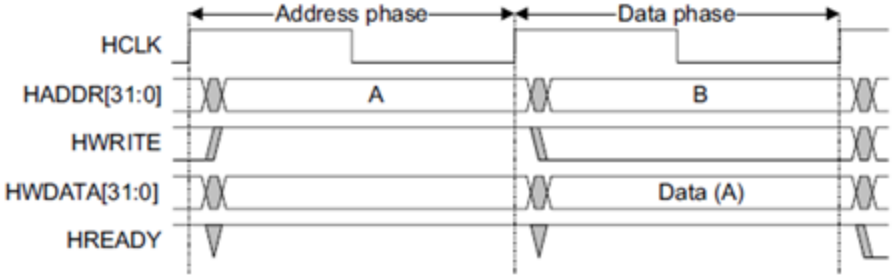
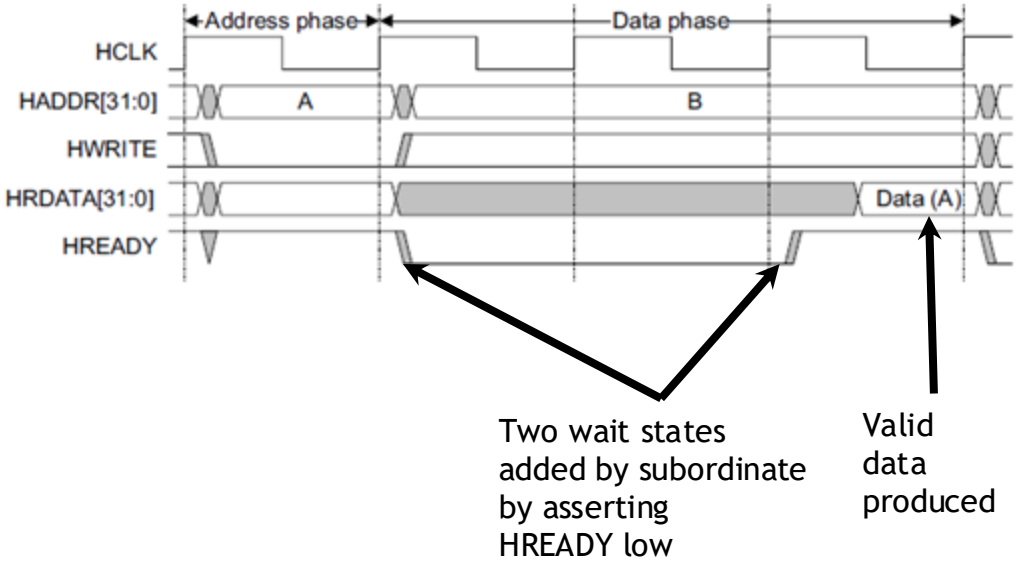
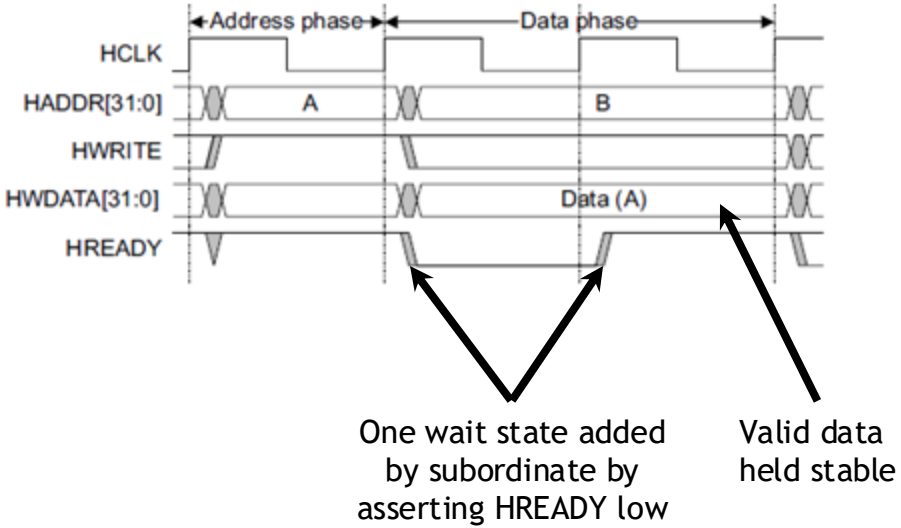


Figure 3-2 Write transfer

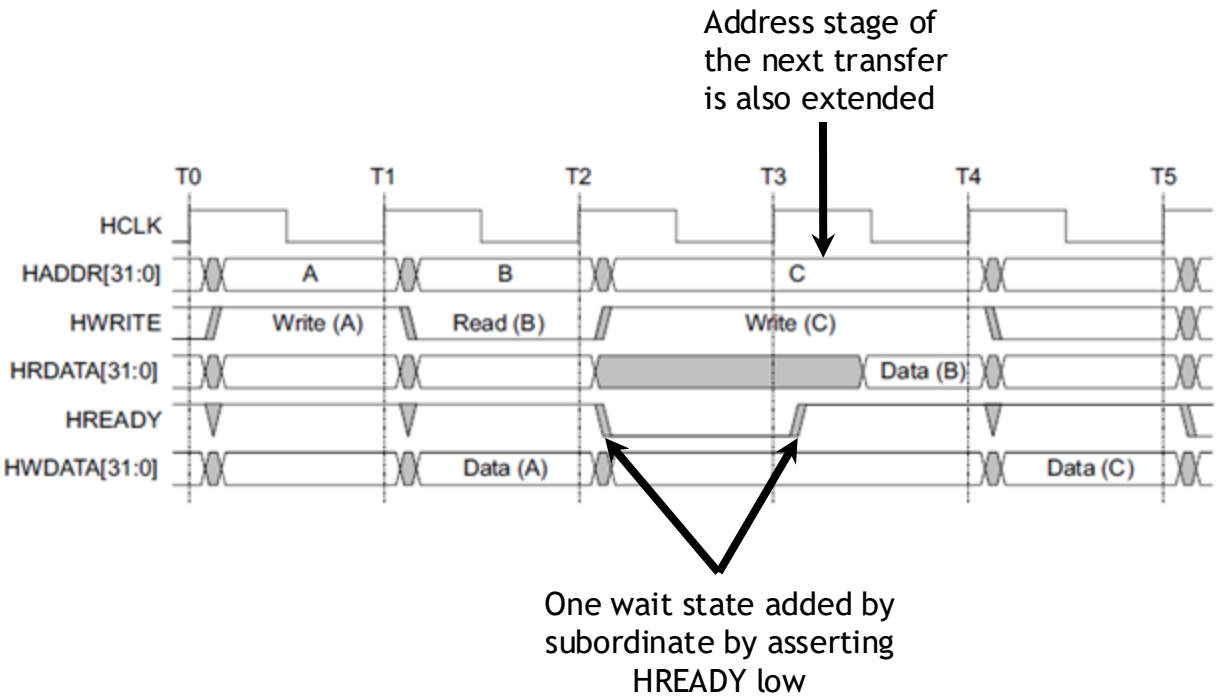
Sometimes the thing you're reading from isn't ready…
Read transfer with two wait states



…or the thing you're writing to isn't ready…
Write transfer with one wait state



Wait states extend the address phase of next transfer



Basic read and write transfers with no wait states

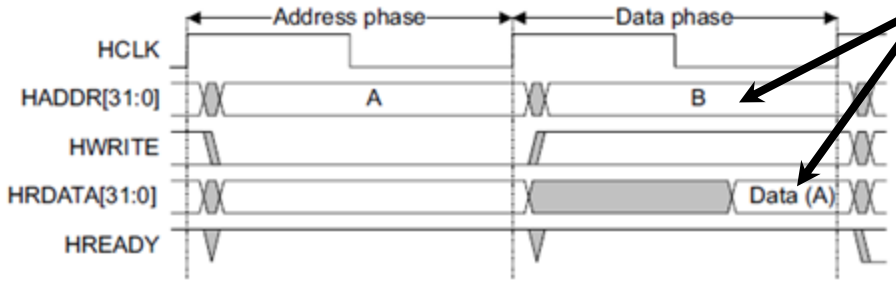


Figure 3-1 Read transfer

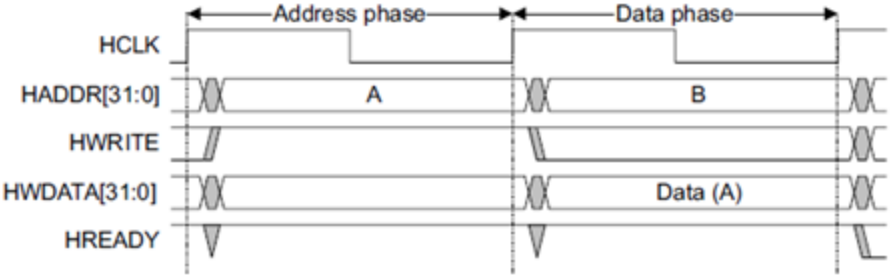
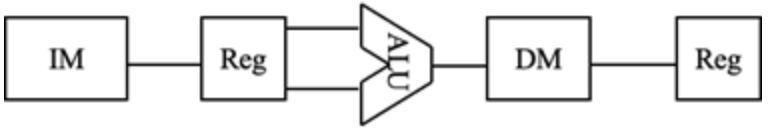
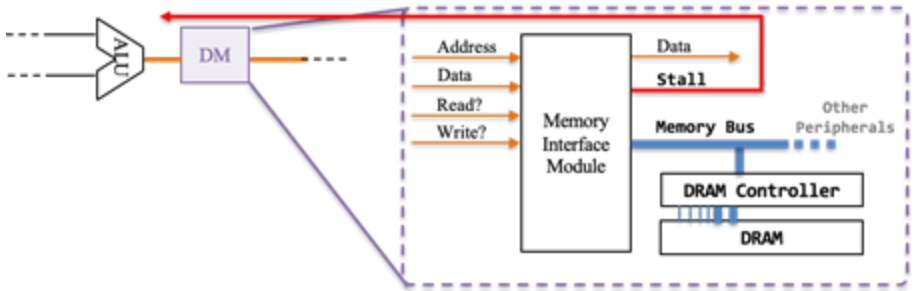


Figure 3-2 Write transfer

Pipelined Address & Data Transfer

What is the implication of this memory bus design for our in-order pipeline?





Example:
ARM Cortex-M0 Technical Reference Manual Instruction Set Summary

Table 3.1. Cortex-M0 instruction summary

Operation	Description	Assembler	Cycles
Move	8-bit immediate	`MOVS Rd, #<imm>`	1
	Lo to Lo	`MOVS Rd, Rm`	1
	Any to Any	`MOV Rd, Rm`	1
	Any to PC	`MOV PC, Rm`	3
Load	Signed byte, register offset	`LDRSB Rd,[Rn, Rm]`	2
	PC-relative	`LDR Rd, <label>`	2
	SP-relative	`LDR Rd,[SP, #<imm>]`	2
	Multiple, excluding base	`LDM Rn!,{<loreglist>}`	1+N[b]
	Multiple, including base	`LDM Rn,{<loreglist>}`	1+N[b]

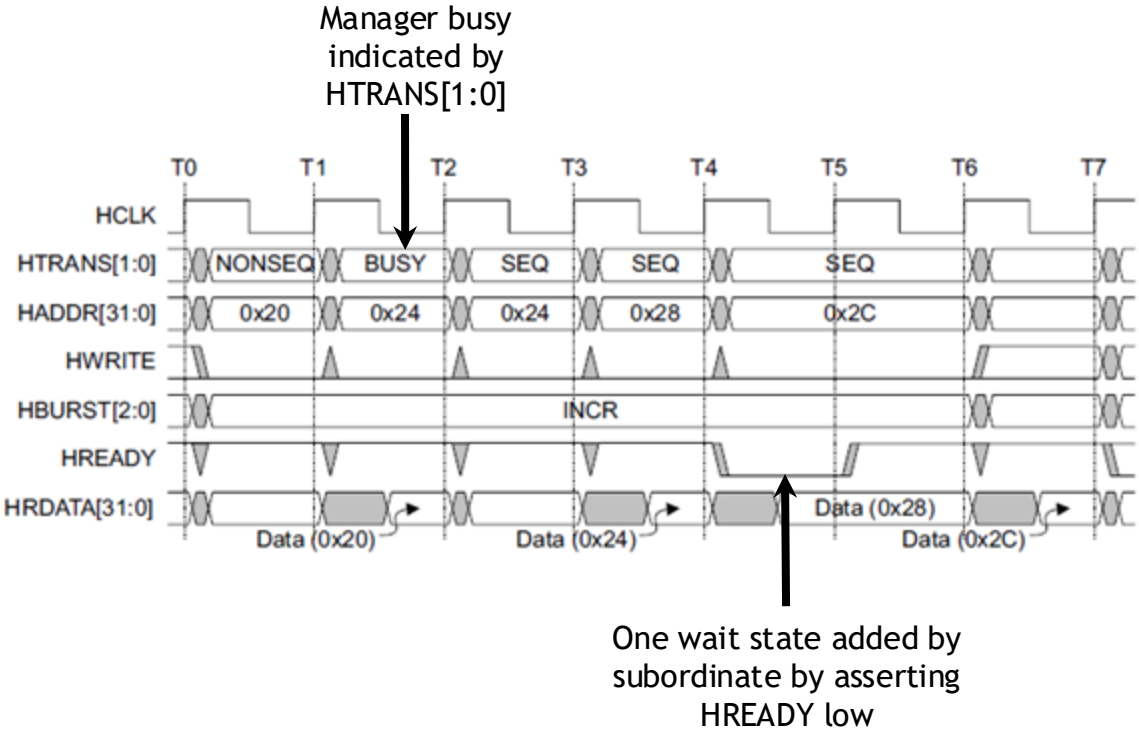
BTW, what's this about?

And what is going on here??

Transfers can be of four types (HTRANS[1:0])

- IDLE (b00)
  - No data transfer is required
  - Subordinate must OKAY w/o waiting
  - Subordinate must ignore IDLE
- BUSY (b01)
  - Insert idle cycles in a burst
  - Burst will continue afterward
  - Address/control reflects next transfer in burst
  - Subordinate must OKAY w/o waiting
  - Subordinate must ignore BUSY
- NONSEQ (b10)
  - Indicates single transfer or first transfer of a burst
  - Address/control unrelated to prior transfers
- SEQ (b11)
  - Remaining transfers in a burst
  - Addr = prior addr + transfer size

A four beat burst with manager busy and subordinate wait



Controlling the size (width) of a transfer

- HSIZE[2:0] encodes the size
- This cannot exceed the data bus width (e.g. 32-bits)
- HSIZE + HBURST is determines wrapping boundary for wrapping bursts
- HSIZE must remain constant throughout a burst transfer

HSIZE[2]	HSIZE[1]	HSIZE[0]	Size (bits)	Description
0	0	0	8	Byte
0	0	1	16	Halfword
0	1	0	32	Word
0	1	1	64	Doubleword
1	0	0	128	4-word line
1	0	1	256	8-word line
1	1	0	512	-
1	1	1	1024	-

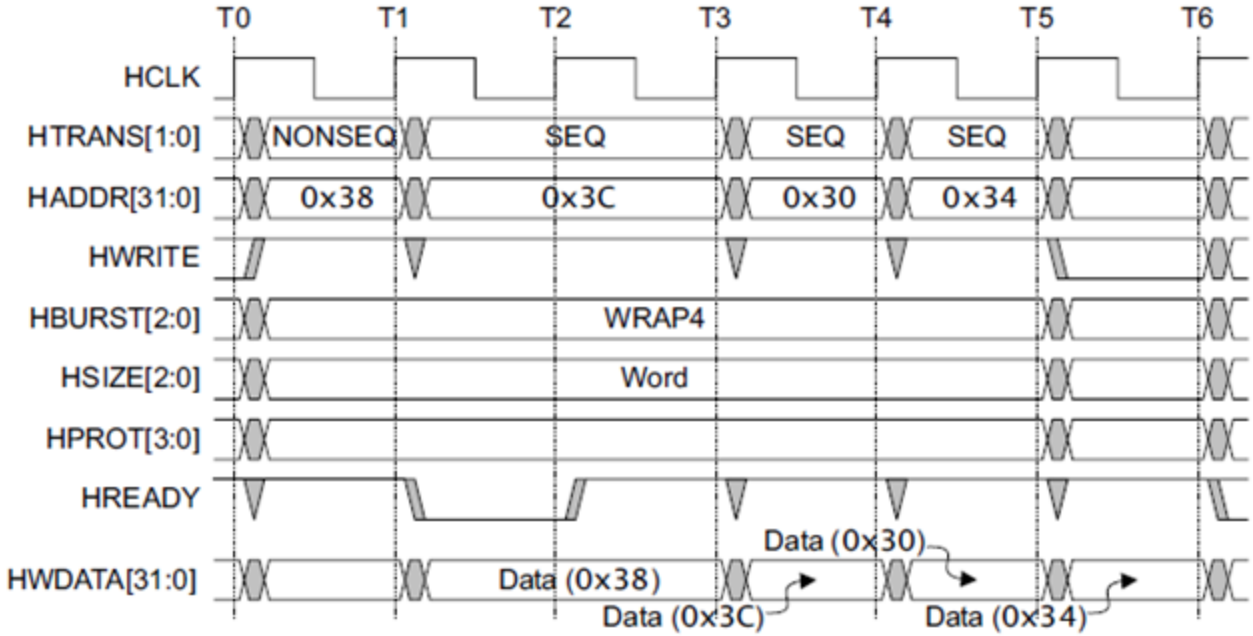
Controlling the burst beats (length) of a transfer

- Burst of 1, 4, 8, 16, and undef
- HBURST[2:0] encodes the type
- Incremental burst
- Wrapping bursts
  - 4 beats x 4-byte words wrapping
  - Wraps at 16 byte boundary
  - E.g. 0x34, 0x38, 0x3c, 0x30,…
- Bursts must not cross 1KB address boundaries

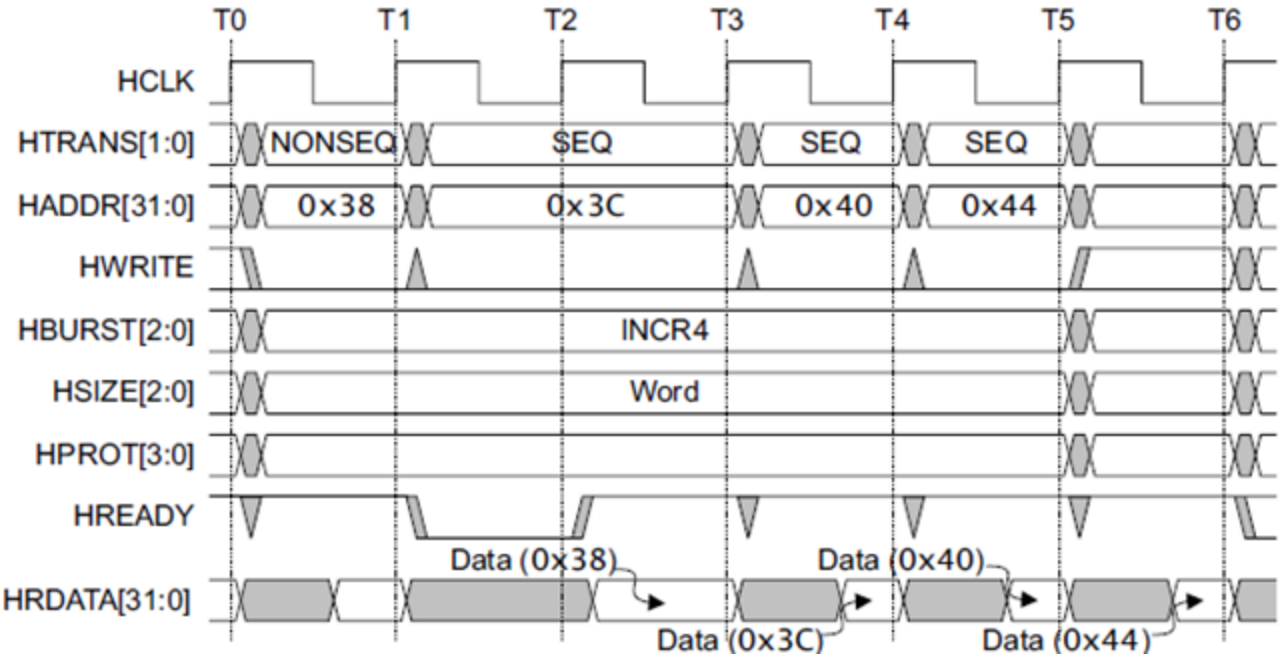
HBURST[2:0]	Type	Description
b000	SINGLE	Single burst
b001	INCR	Incrementing burst of undefined length
b010	WRAP4	4-beat wrapping burst
b011	INCR4	4-beat incrementing burst
b100	WRAP8	8-beat wrapping burst
b101	INCR8	8-beat incrementing burst
b110	WRAP16	16-beat wrapping burst
b111	INCR16	16-beat incrementing burst

Many slides in this subsection borrowed from Prabal

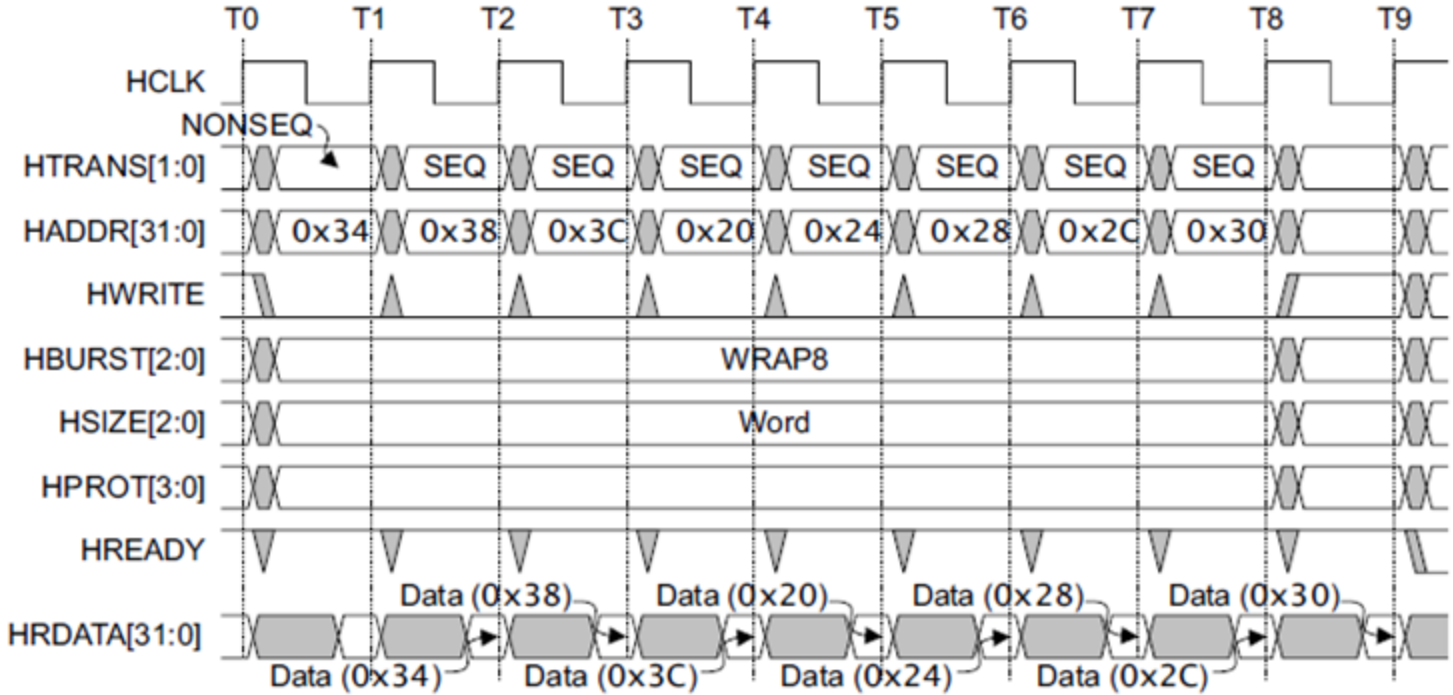
A four beat wrapping burst (WRAP4)



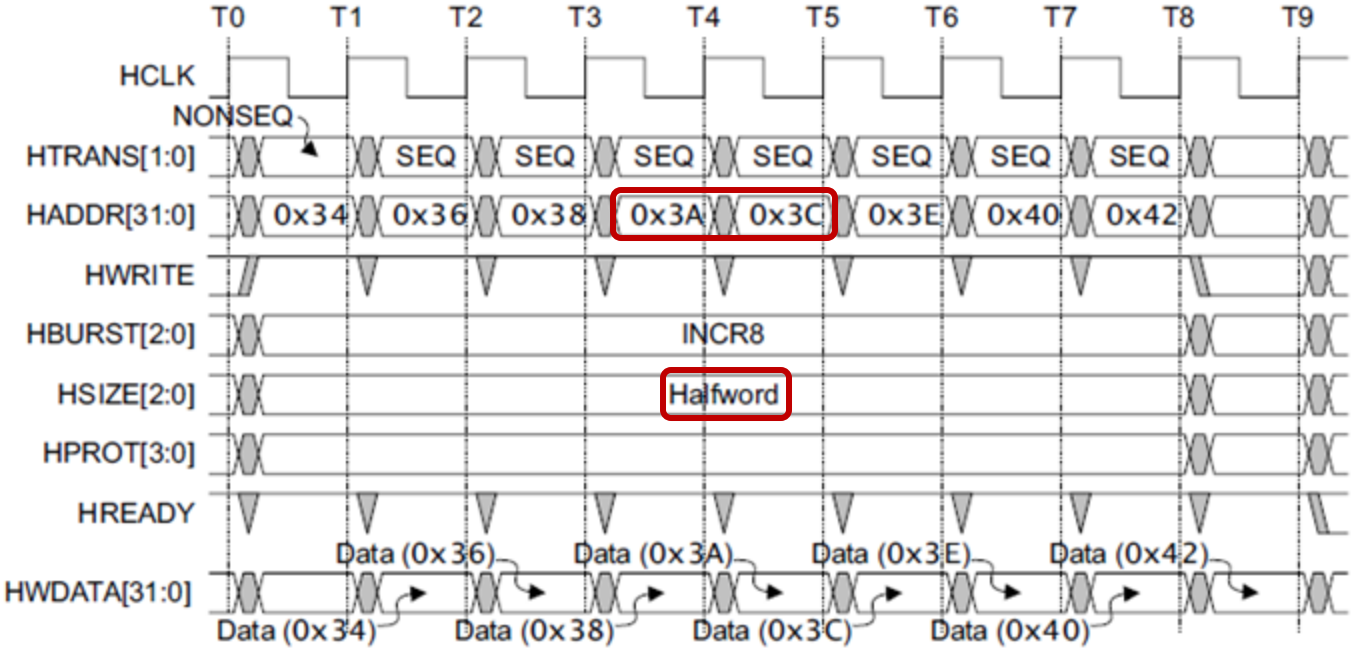
A four beat incrementing burst (INCR4)



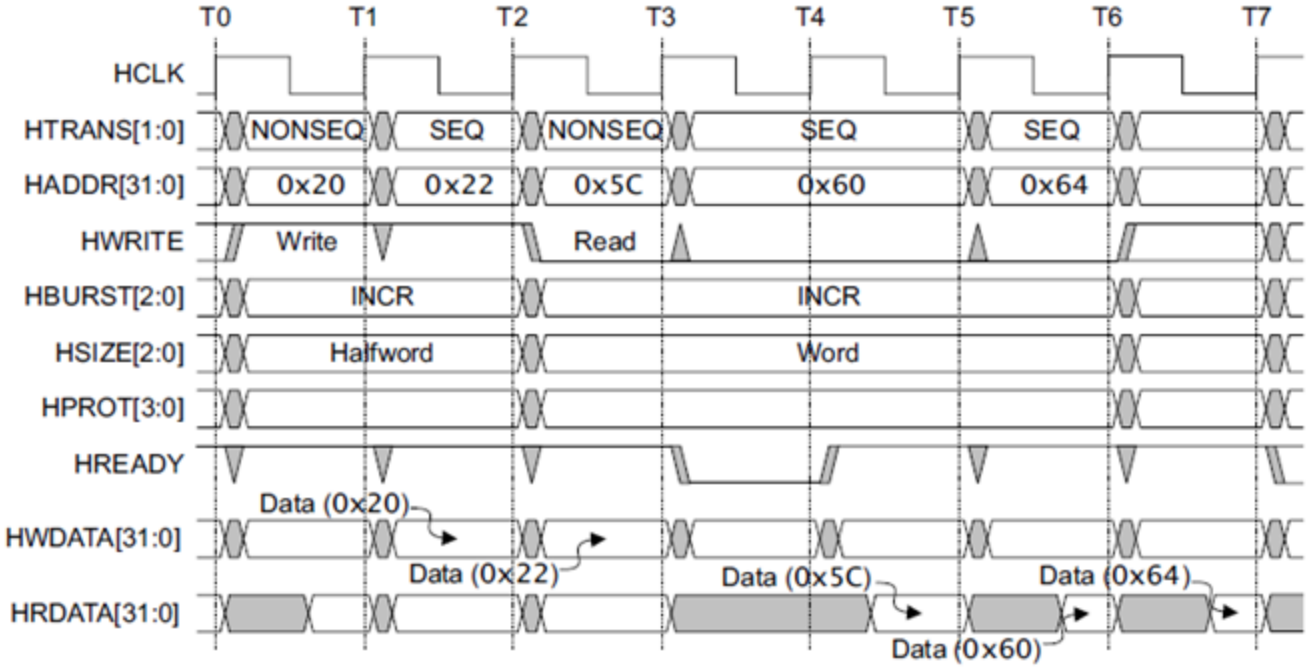
An eight beat wrapping burst (WRAP8)



An eight beat incrementing burst (INCR8) using half-word transfers

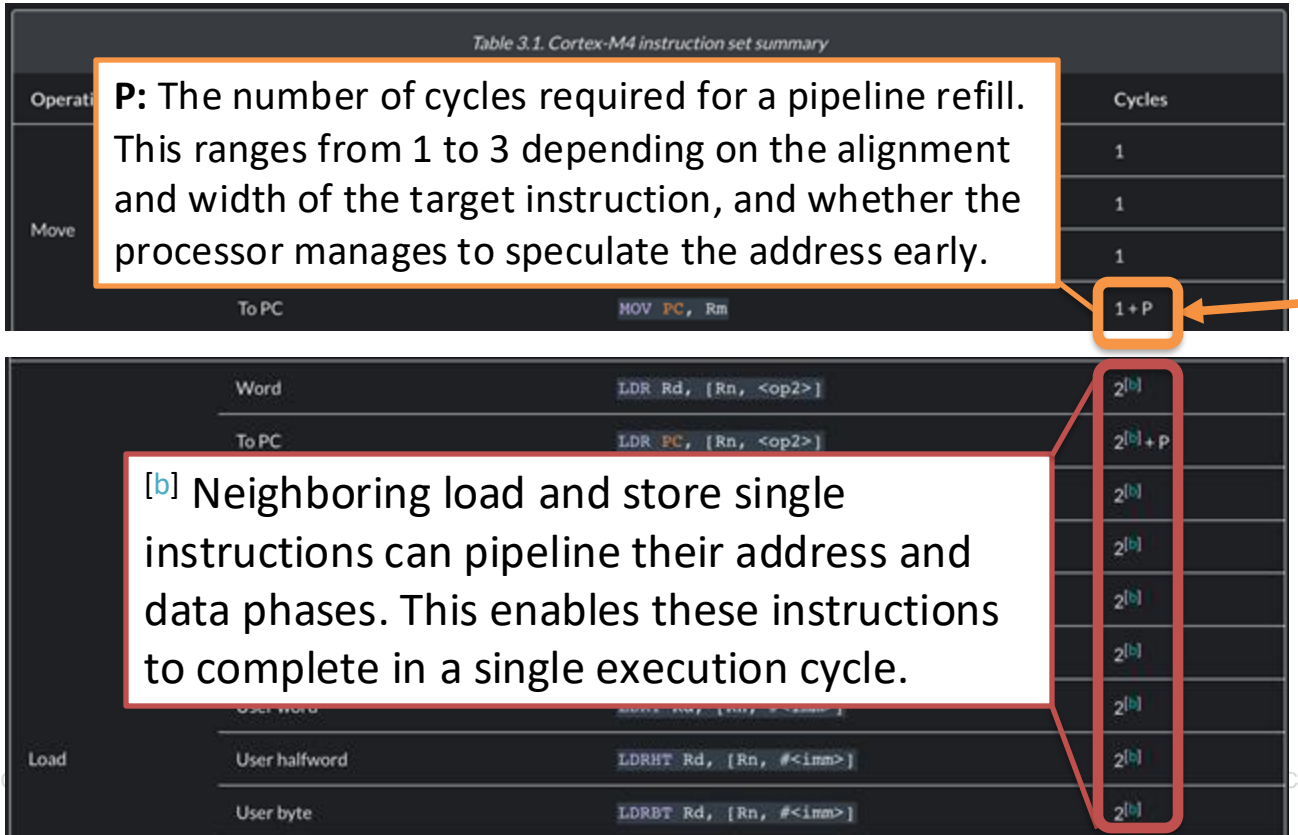


An undefined length incrementing burst (INCR)



What uses all this crazy stuff??

Example: ARM Cortex-M4 Technical Reference Manual Instruction Set Summary



P: The number of cycles required for a pipeline refill. This ranges from 1 to 3 depending on the alignment and width of the target instruction, and whether the processor manages to speculate the address early.

[b] Neighboring load and store single instructions can pipeline their address and data phases. This enables these instructions to complete in a single execution cycle.

BTW, what's this about?

(It was just "3" for the simpler Cortex-M0…)

The Point: Even on “simple” microcontrollers, memory interfaces are complicated for performance reasons

- **Key Concepts**
  - Not everything behind a memory address is actually ‘memory’
  - Memory bus reads and writes may have unpredictable latency
    - And in-order pipelines must stall all execution until memory resolves
  - Memory accesses need to run roughly at the same speed as the CPU for reasonable performance

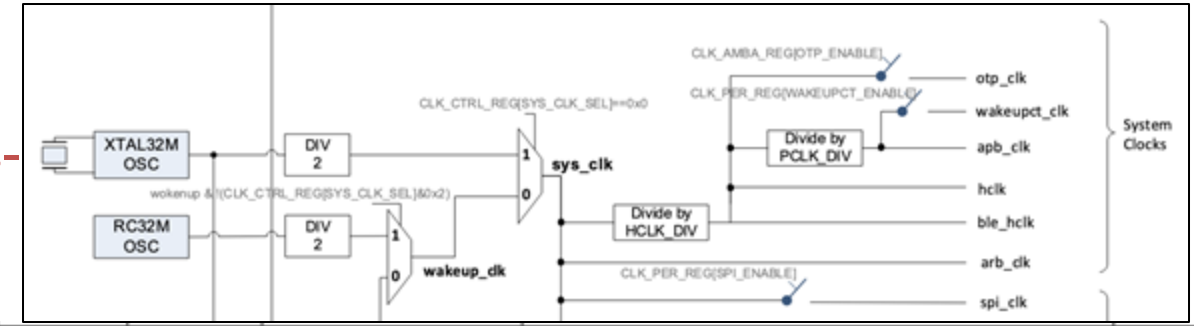
MEMORY & CACHE DESIGNS IN APPLICATIONS PROCESSORS

Finally, telling the truth about Memory

- Before today, we've assumed memory can be accessed in a single cycle
  - Recall: (Almost) still true for embedded CPUs!

What controls memory bus speed anyway?

- Usually the memory bus clock is derived from the system bus clock
  - Microcontrollers are slow enough that memory can (mostly) “keep up”
- But not always…
  - E.g., DA14530
- No point in “always-waiting” if every access would add wait states



1:0	R/W	HCLK_DIV	AHB interface and microprocessor clock (HCLK). HCLK is source clock divided by: 0x0: divide by 1 0x1: divide by 2 0x2: divide by 4 0x3: divide by 8	0x0

Finally, telling the truth about Memory

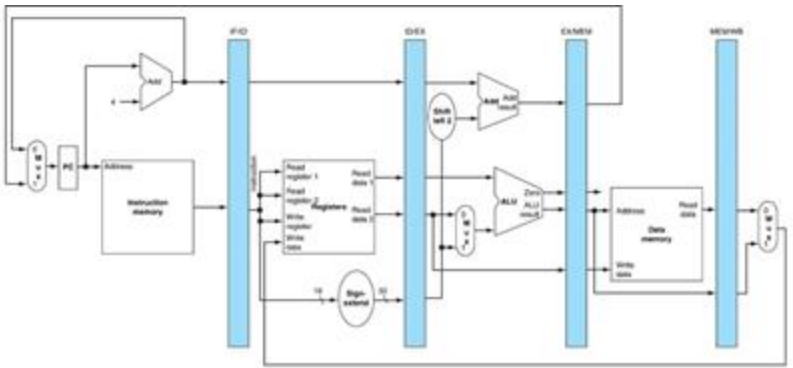
- Before today, we've assumed memory can be accessed in a single cycle
  - Recall: (Almost) still true for embedded CPUs!
- For "Applications Processors" (high performance machines)…
  - Cycle time has decreased rapidly
  - Memory access time has decreased very little
  - Result: **Memory latency can be in the neighborhood of 350-500 cycles!**

The truth about memory latency

- So then what is the point of pipelining, branch prediction, etc. if memory latency is 500 cycles?
- Keep in mind, 20% of instructions are loads and stores, and we fetch (read inst memory) every instruction.

```
lw  R4, 1000(R2)   IF ID EX M ---------  ...  --------WB
lw  R8, 200(R4)      IF  ID  B ---------       --------ID EX M--------... ------ ----  WB
add  R10, R8, R10      IF  B ---------       --------    ID  B ---------       --------ID EX M
```

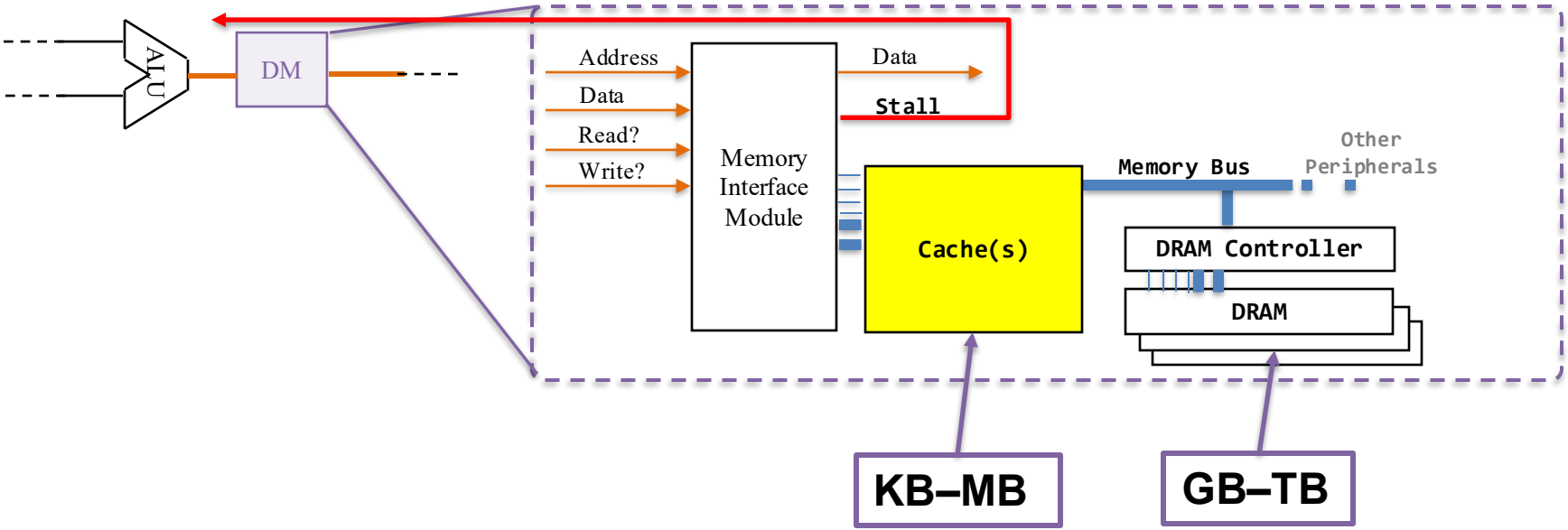
CPI = ~ ??



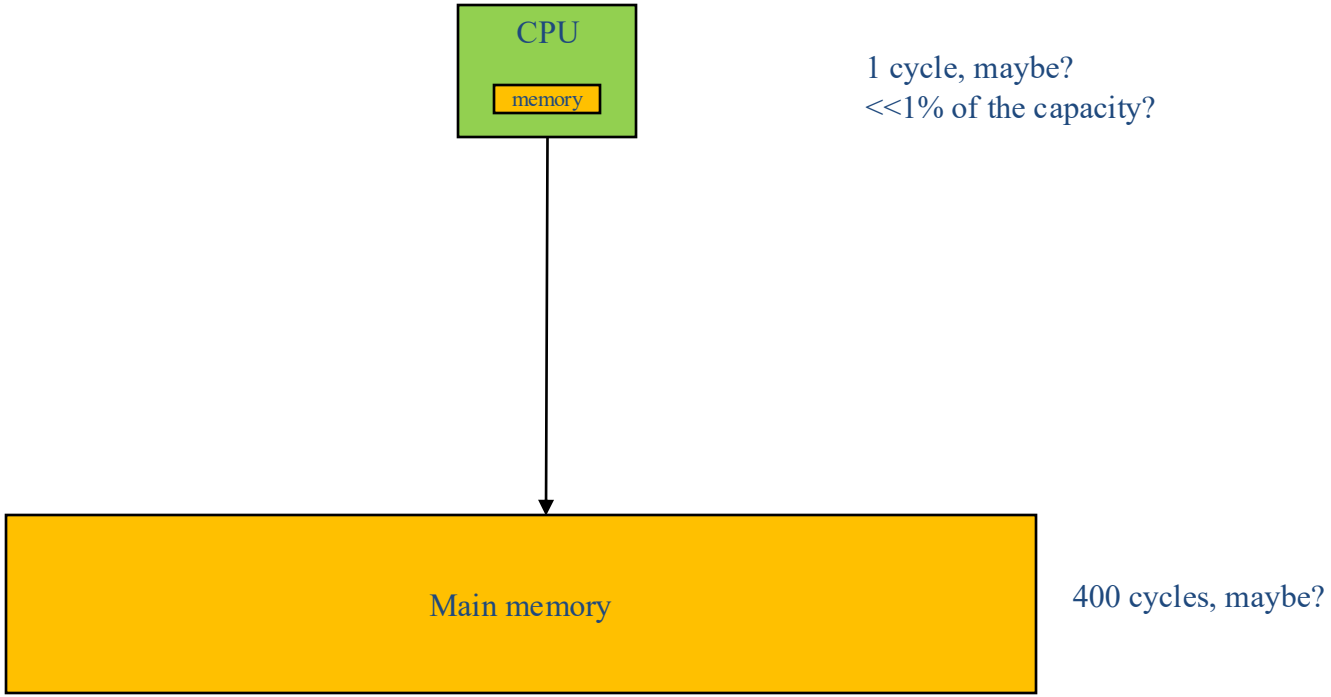
But wait…

- That is assuming DRAM technology…
  - which is necessary for large main memories (multiple gigabytes, for example)
- But we can design much smaller (capacity) memories using SRAM
  - even on chip!
- If we still want to access it in a cycle, it should be KB, not MB or GB
  - (Indeed, microcontrollers usually have <= 1MB of SRAM and no DRAM)

Let's put one of these fast, "tiny-memories" near the CPU!

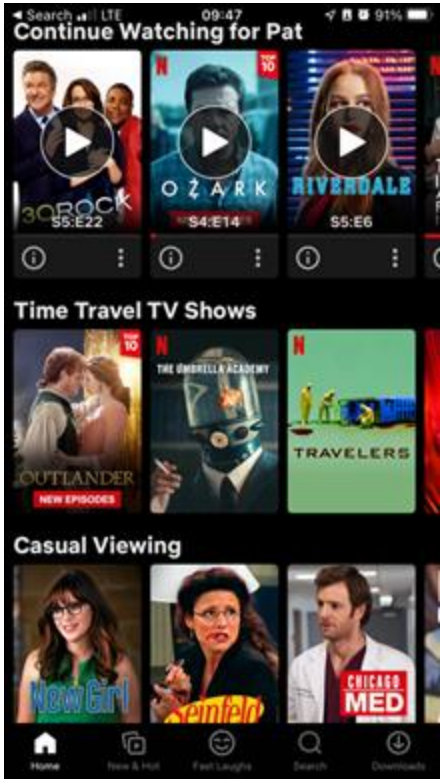


So what can I do with this?



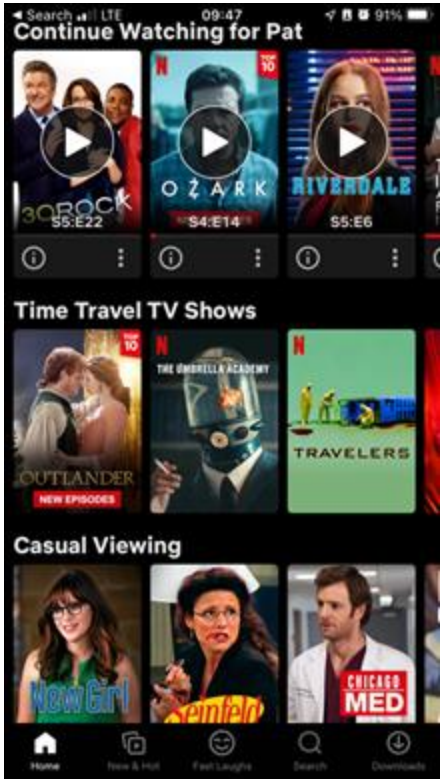
Memory Locality

- Memory hierarchies take advantage of ***memory locality***.



Memory Locality

- Memory hierarchies take advantage of ***memory locality***.
- *Memory locality* is the principle that future memory accesses are *near* past accesses.



Memory Locality

- Memory hierarchies take advantage of memory locality.
- Memory locality is the principle that future memory accesses are near past accesses.
- Memories take advantage of two types of locality
  - near in time => we will often access the same data again very soon
  - near in space/distance => our next access is often very close to our last access (or recent accesses).

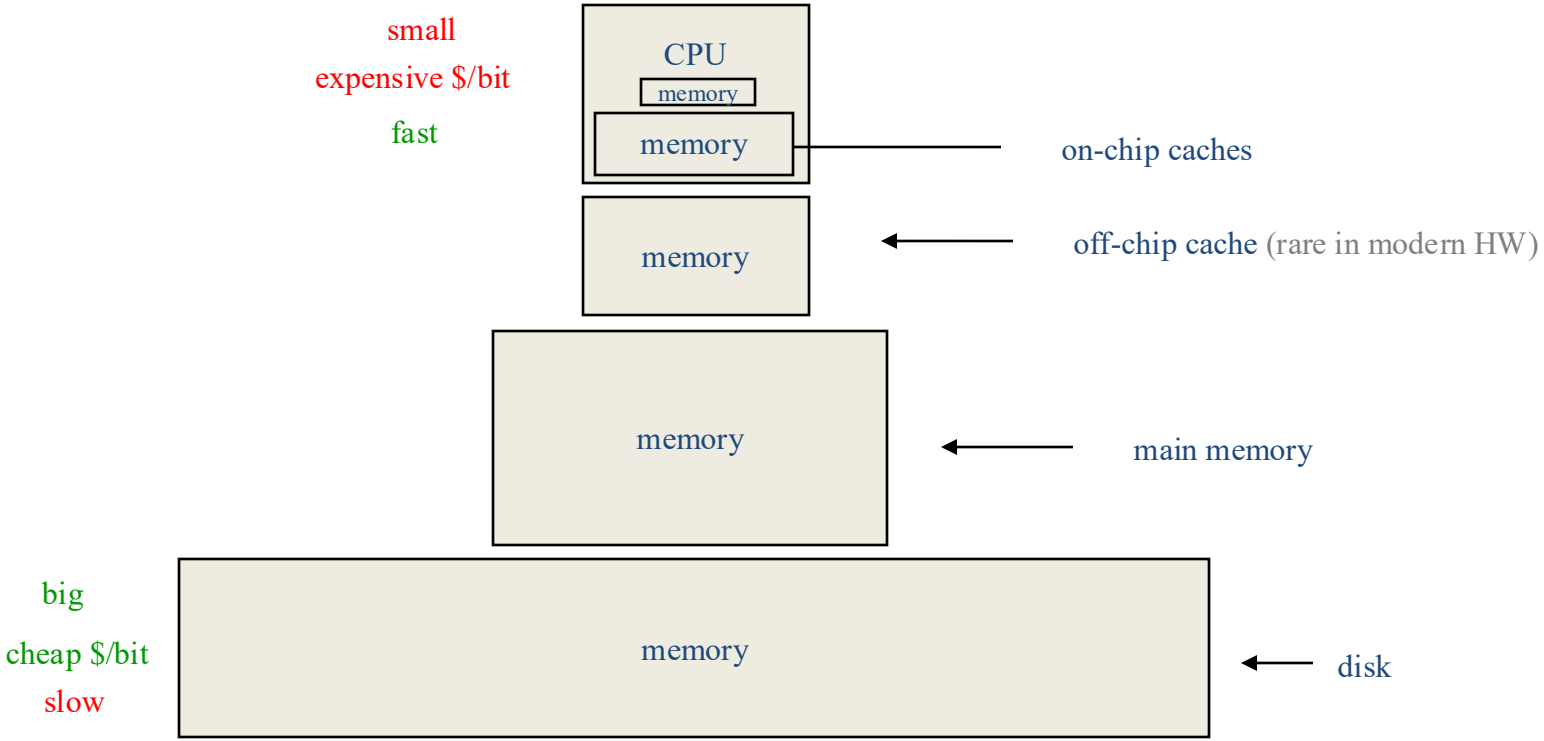
- (this sequence of addresses exhibits both temporal and spatial locality)
  - 1,2,3,1,2,3,8,8,47,9,10,8,8...

Locality and cacheing

- Memory hierarchies exploit locality by cacheing (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can't build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity

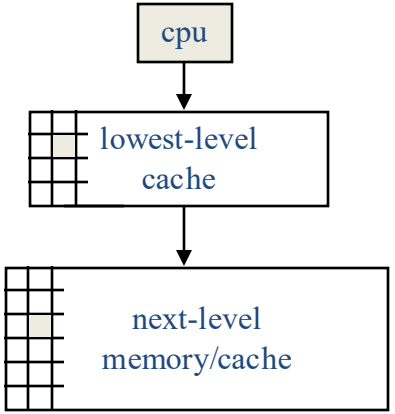
- SRAM access times are ~1ns at cost of $2000 to $5000 per Gbyte.
- DRAM access times are ~70ns at cost of $20 to $75 per Gbyte.
- Disk access times are 5 to 20 million ns at cost of $.20 to $2 per Gbyte.

A typical memory hierarchy



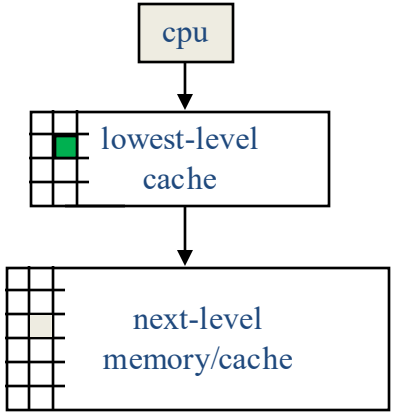
so then where is my program and data??

Cache Fundamentals



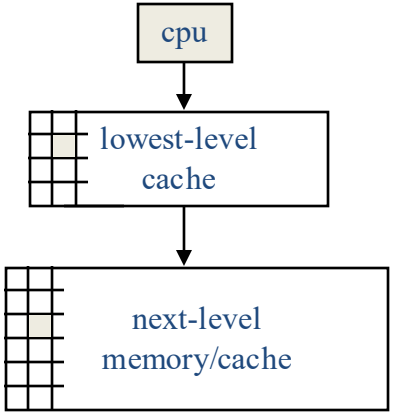
Cache Fundamentals

- ***cache hit*** -- an access where the data is found in the cache.



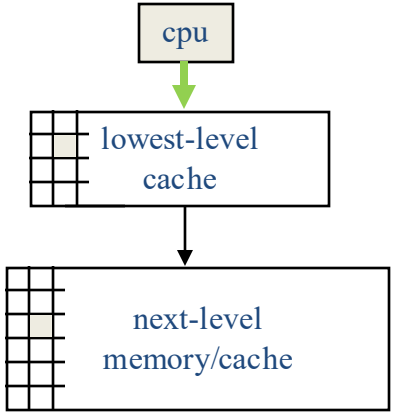
Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- ***cache miss*** -- an access which isn't



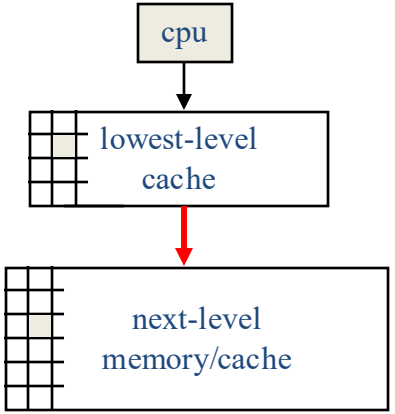
Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- ***hit time*** -- time to access the cache



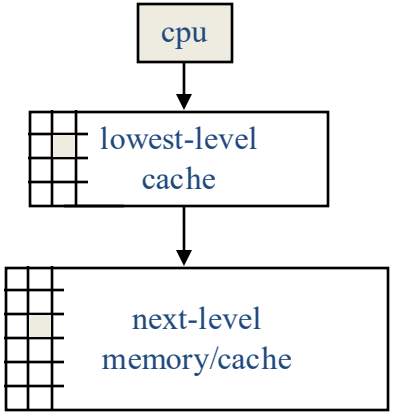
Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- ***miss penalty*** -- time to move data from further level to closer

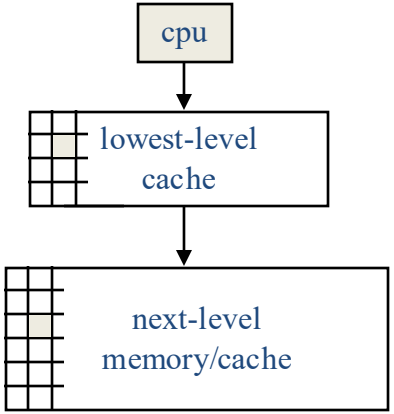


Cache Fundamentals

- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- miss penalty -- time to move data from further level to closer
- ***hit ratio*** -- percentage of time the data is found in the cache



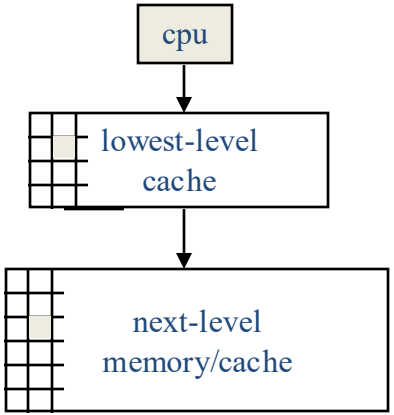
Cache Fundamentals



- cache hit -- an access where the data is found in the cache.
- cache miss -- an access which isn't
- hit time -- time to access the cache
- miss penalty -- time to move data from further level to closer
- hit ratio -- percentage of time the data is found in the cache
- ***miss ratio*** -- (1 - hit ratio)

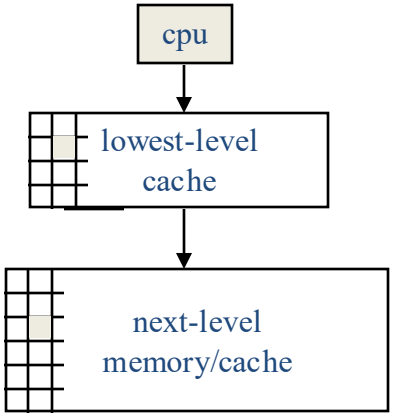
Cache Fundamentals, cont.

- ***cache block size*** or ***cache line size***– the amount of data that gets transferred on a cache miss.



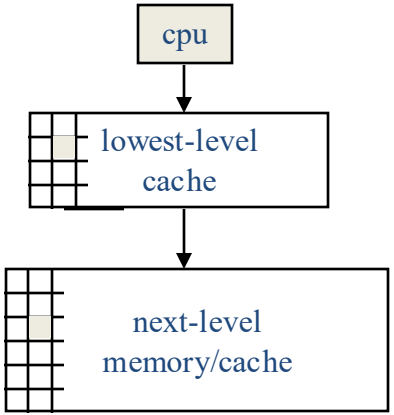
Cache Fundamentals, cont.

- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- ***instruction cache*** – cache that only holds instructions.



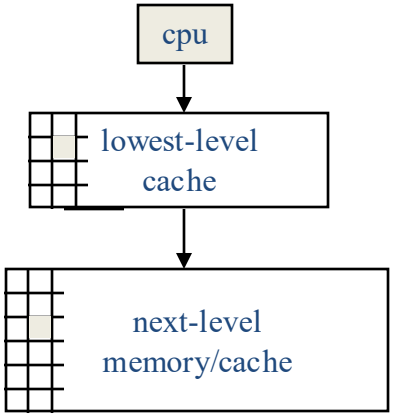
Cache Fundamentals, cont.

- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- *instruction cache* – cache that only holds instructions.
- ***data cache*** – cache that only caches data.



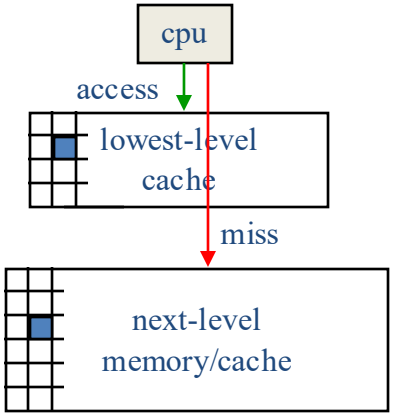
Cache Fundamentals, cont.

- *cache block size* or *cache line size* – the amount of data that gets transferred on a cache miss.
- *instruction cache* – cache that only holds instructions.
- *data cache* – cache that only caches data.
- ***unified cache*** – cache that holds both.



Cacheing Issues

- On a memory access -
  - How do I know if this is a hit or miss?
- On a cache miss -
  - where to put the new data?
  - what data to throw out?
  - how to remember what data this is?



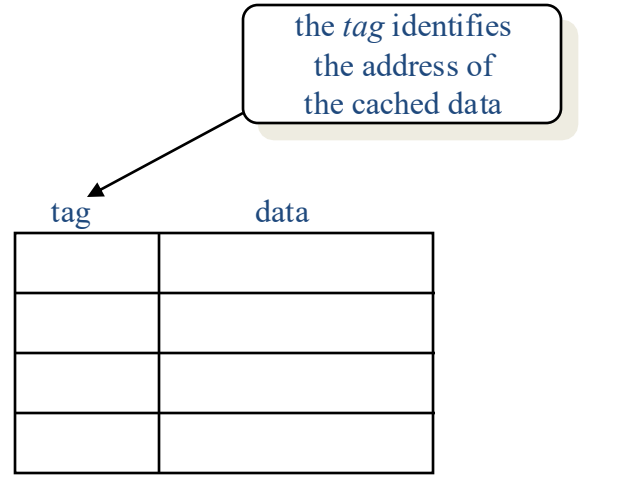
Hardware implications on cache design

- Caches are basically *the* thing that make real workloads fast
- The size of a cache is inversely proportional to its speed
  - Smaller caches are faster
- And **every bit counts**

- This is why caches use as few **bits** as possible to do their work
  - This makes caches tricky to walk through as a human

A simple cache

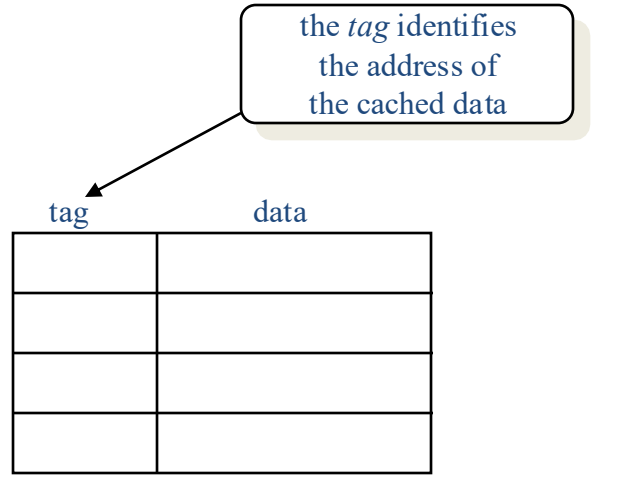
```
address string:
4           00000100
8           00001000
12          00001100
4           00000100
8           00001000
20          00010100
4           00000100
8           00001000
20          00010100
24          00011000
12          00001100
8           00001000
4           00000100
```



- A cache that can put a line of data anywhere is called _____
- The most popular replacement strategy is *LRU* ( ).

A simple cache

address string:	
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100

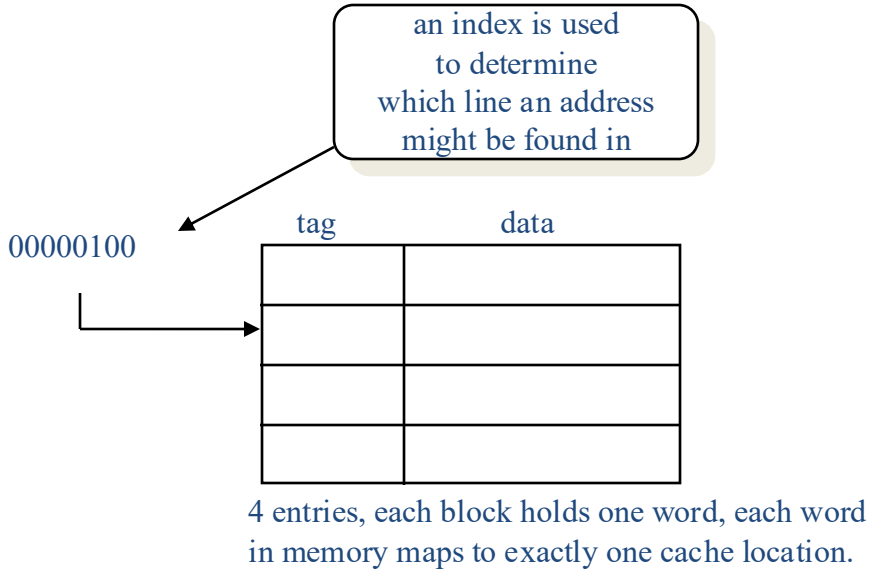


4 entries, each block holds one word, any block can hold any word.

- A cache that can put a line of data anywhere is called **Fully Associative**
- The most popular replacement strategy is *LRU* (**Least Recently Used**).

A simpler cache

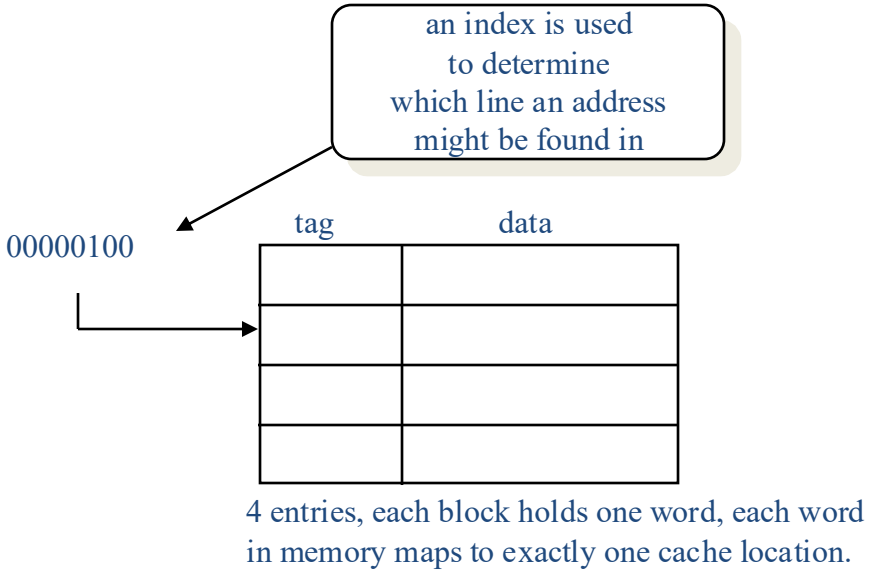
address string:	
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



- A cache that can put a line of data in exactly one place is called _____.
- Advantages/disadvantages vs. fully-associative?

A simpler cache

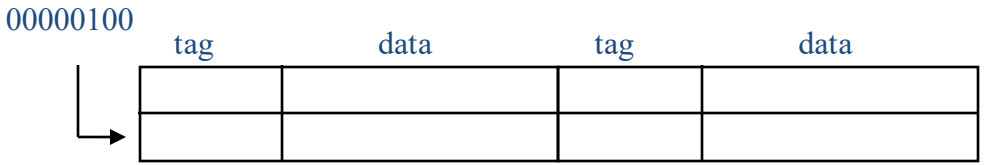
address	string:
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



- A cache that can put a line of data in exactly one place is called **direct mapped**
- Advantages/disadvantages vs. fully-associative?

A set-associative cache

```
address string:
4           00000100
8           00001000
12          00001100
4           00000100
8           00001000
20          00010100
4           00000100
8           00001000
20          00010100
24          00011000
12          00001100
8           00001000
4           00000100
```



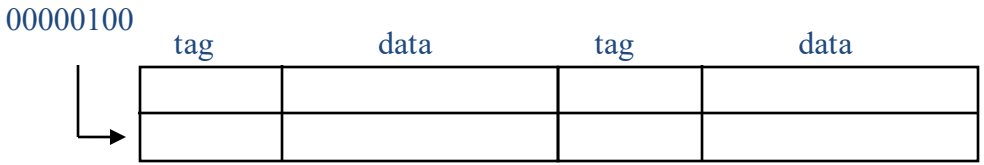
4 entries, each block holds one word, each word in memory maps to one of a set of *n* cache lines

- A cache that can put a line of data in exactly *n* places is called *n-way* _____.
- The cache lines/blocks that share the same index are a cache _____.

A set-associative cache

address string:

4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



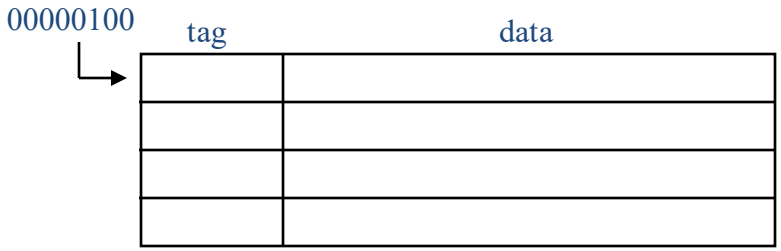
4 entries, each block holds one word, each word in memory maps to one of a set of *n* cache lines

- A cache that can put a line of data in exactly *n* places is called ***n-way set-associative***.
- The cache lines/blocks that share the same index are a cache **set**.

Longer Cache Blocks

address string:

4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



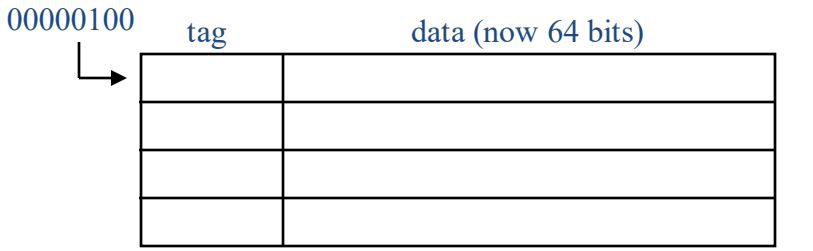
4 entries, each block holds two words, each word in memory maps to exactly one cache location (this cache is twice the total size of the prior caches).

- Large cache blocks take advantage of *spatial locality*.
- Too large of a block size can waste cache space.
- Longer cache blocks require less tag space

Longer Cache Blocks

address string:

4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



4 entries, each block holds two words, each word in memory maps to exactly one cache location (this cache is twice the total size of the prior caches).

- Large cache blocks take advantage of ***spatial locality***.
- Too large of a block size can waste cache space.
- Longer cache blocks require less tag space

Q: Describing Cache Type Tradeoffs?

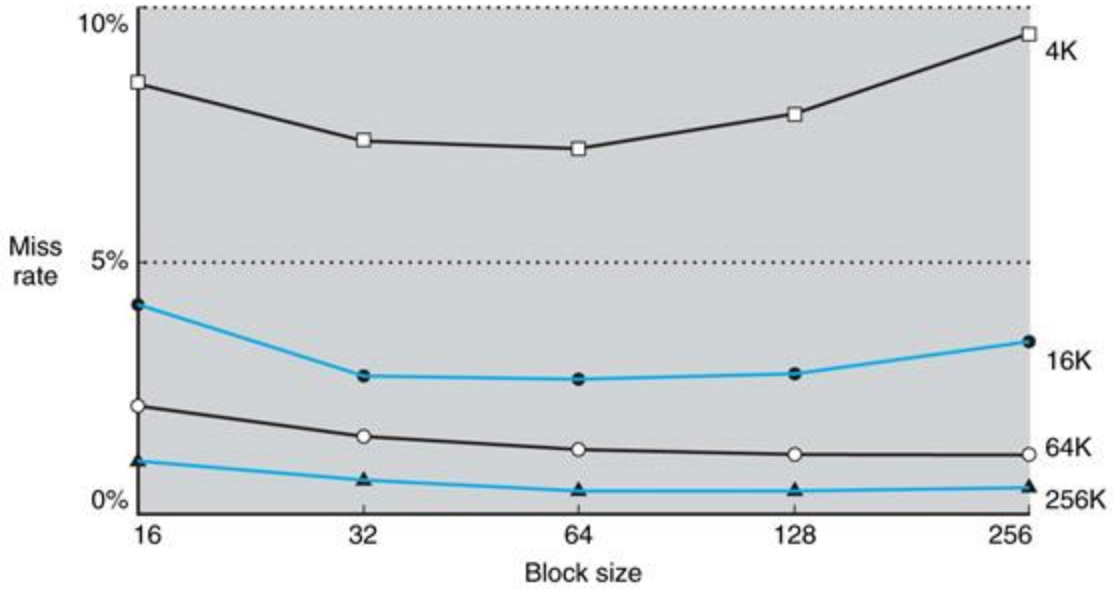
1. Exceptional usage of the cache space in exchange for a slow hit time
2. Poor usage of the cache space in exchange for an excellent hit time
3. Reasonable usage of cache space in exchange for a reasonable hit time

Selection	Fully-Associative	4-way Set Associative	Direct Mapped
A	3	2	1
B	3	3	2
C	1	2	3
D	3	2	1
E	None of the above		

Back to Block Size

- If block size increases spatial locality, should we just make the cache block size really, really big????

Block Size and Miss Rate



Cache Parameters

Cache size = Number of sets * block size * associativity

Warning / Notice—Things that count towards "cache size": cache **data**

Things that **do not count** towards "cache size": tags, valid bits, etc…

Cache Parameters

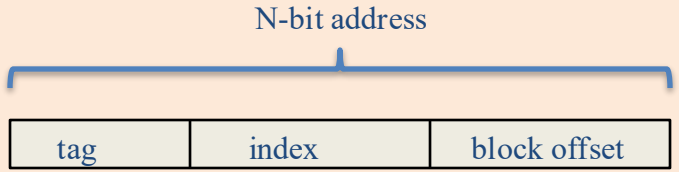
Cache size = Number of sets * block size * associativity

- 128 blocks, 32-byte block size, direct mapped, size = ?

- 128 KB cache, 64-byte blocks, 512 sets, associativity = ?

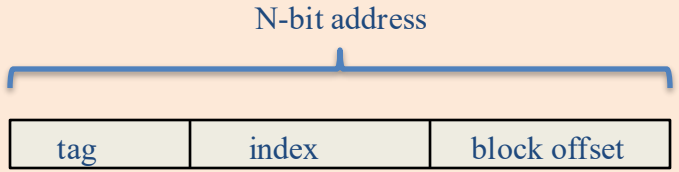
(always keep in mind “cache size” only counts the data storage)

Q: How many bits for each field?



- Generally, we have variables **block_size, cache_size,** and **memory_size**
  - Let's work it out for
    - BS = 8 bytes
    - CS = 1 KB
    - MS = 4 MB
  - And we have a 4-way set-associative cache?
- **What is the…**
  - Number of bits for the block offset: ____
  - Number of bits for the index: ____
  - Number of bits for the tag: ____

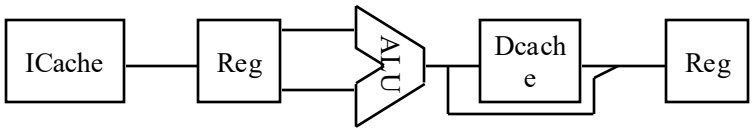
Q: How many bits for each field?



- Generally, we have variables **block_size, cache_size,** and **memory_size**
  - Let's work it out for
    - BS = 32 bytes
    - CS = 16 KB
    - MS = 8 MB
  - And we have a 8-way set-associative cache?
- **What is the…**
  - Number of bits for the block offset: ____
  - Number of bits for the index: ____
  - Number of bits for the tag: ____

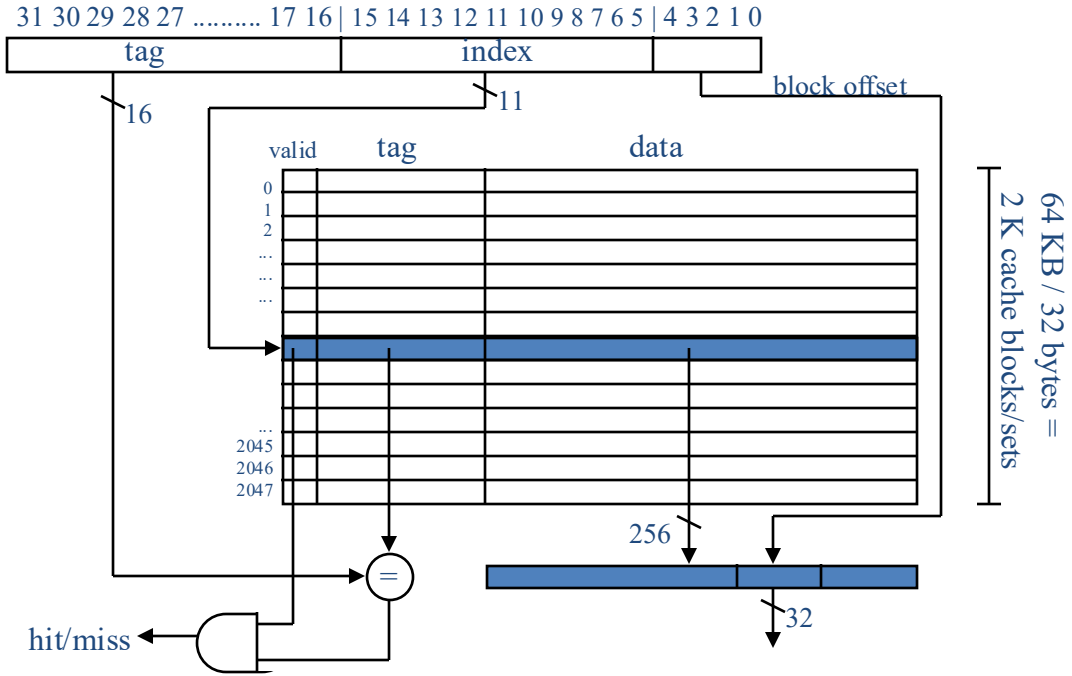
Handling a Cache Access

- 1. Use index and tag to access cache and determine hit/miss.
- 2. If hit, return requested data.
- 3. If miss, select a cache block to be replaced, and access memory or next lower cache (possibly stalling the processor).
  - load entire missed cache line into cache
  - return requested data to CPU (or higher cache)
- 4. If next lower memory is a cache, goto step 1 for that cache.



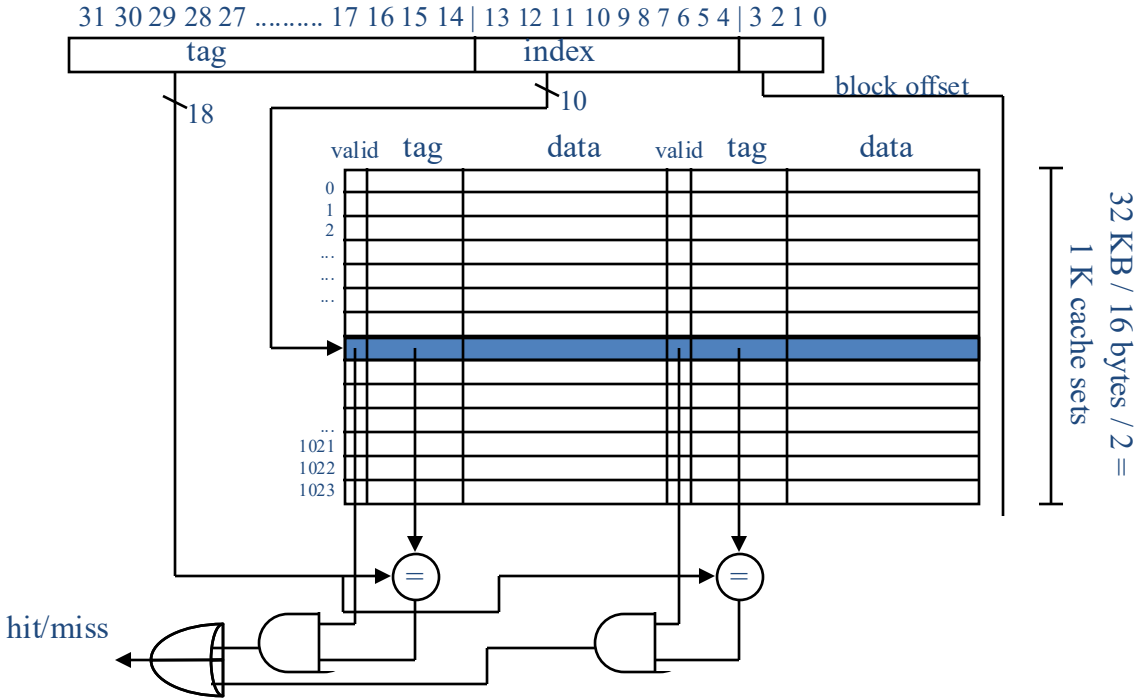
Accessing a Sample Cache

- 64 KB cache, direct-mapped, 32-byte cache block size

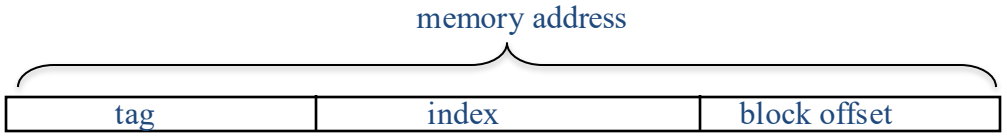


Accessing a Sample Cache

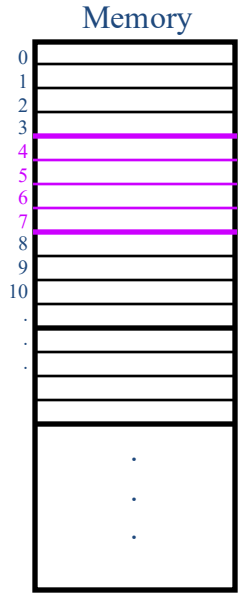
- 32 KB cache, 2-way set-associative, 16-byte block size



Cache Alignment

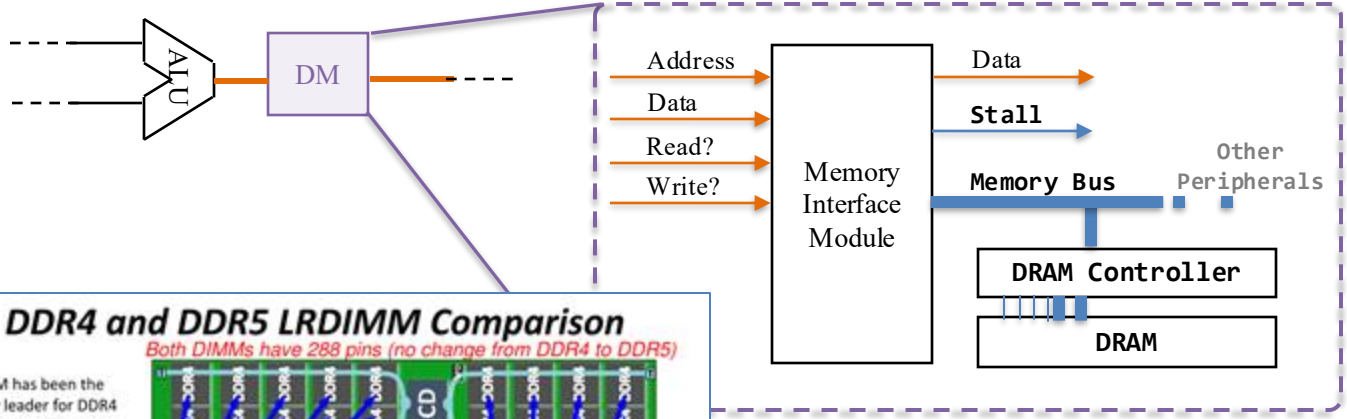


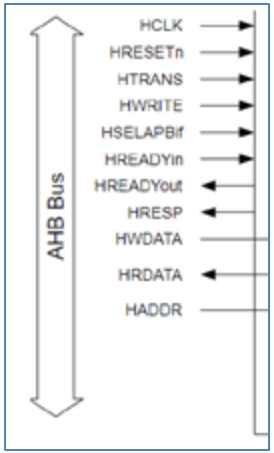
- The data that gets moved into the cache on a miss are all data whose addresses share the same tag and index (regardless of which data gets accessed first).
- This results in
  - no overlap of cache lines
  - easy mapping of addresses to cache lines (no additions)
  - data at address X always being present in the same location in the cache block (at byte X `mod` blocksize) if it is there at all.
- Think of main memory as organized into cache-line sized pieces (because in reality, it is!).

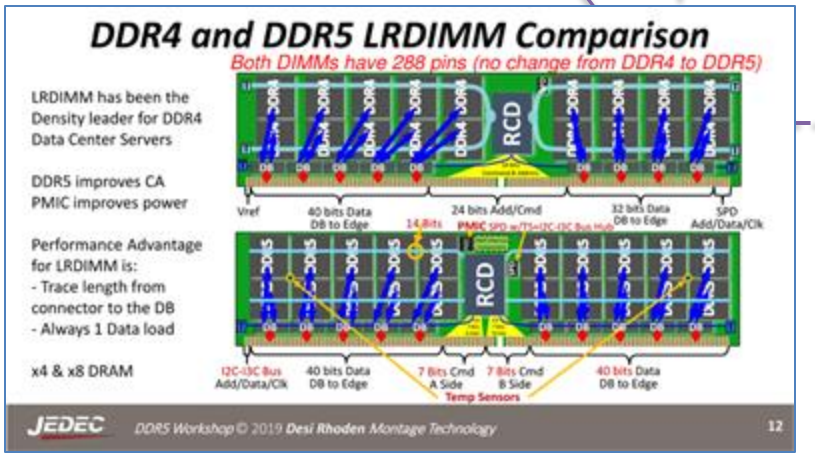


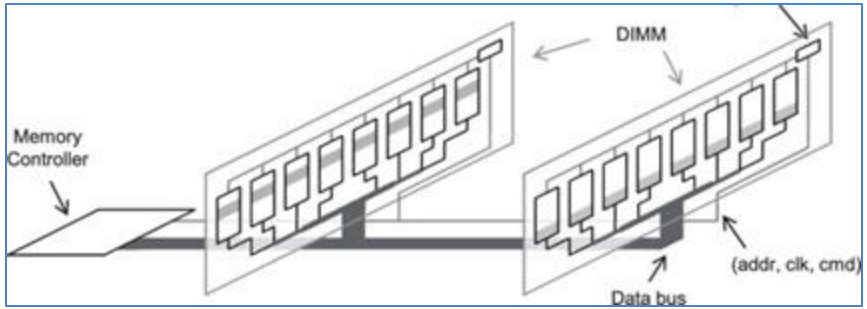
How does the cache actually connect to the pipeline?

[Simplified View! Very different between embedded & high-perf!]



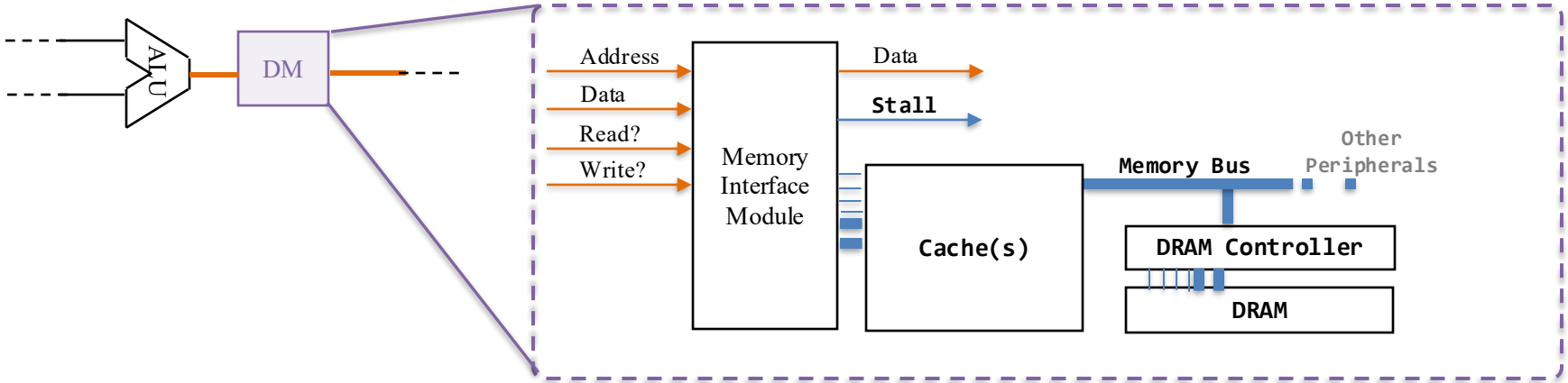






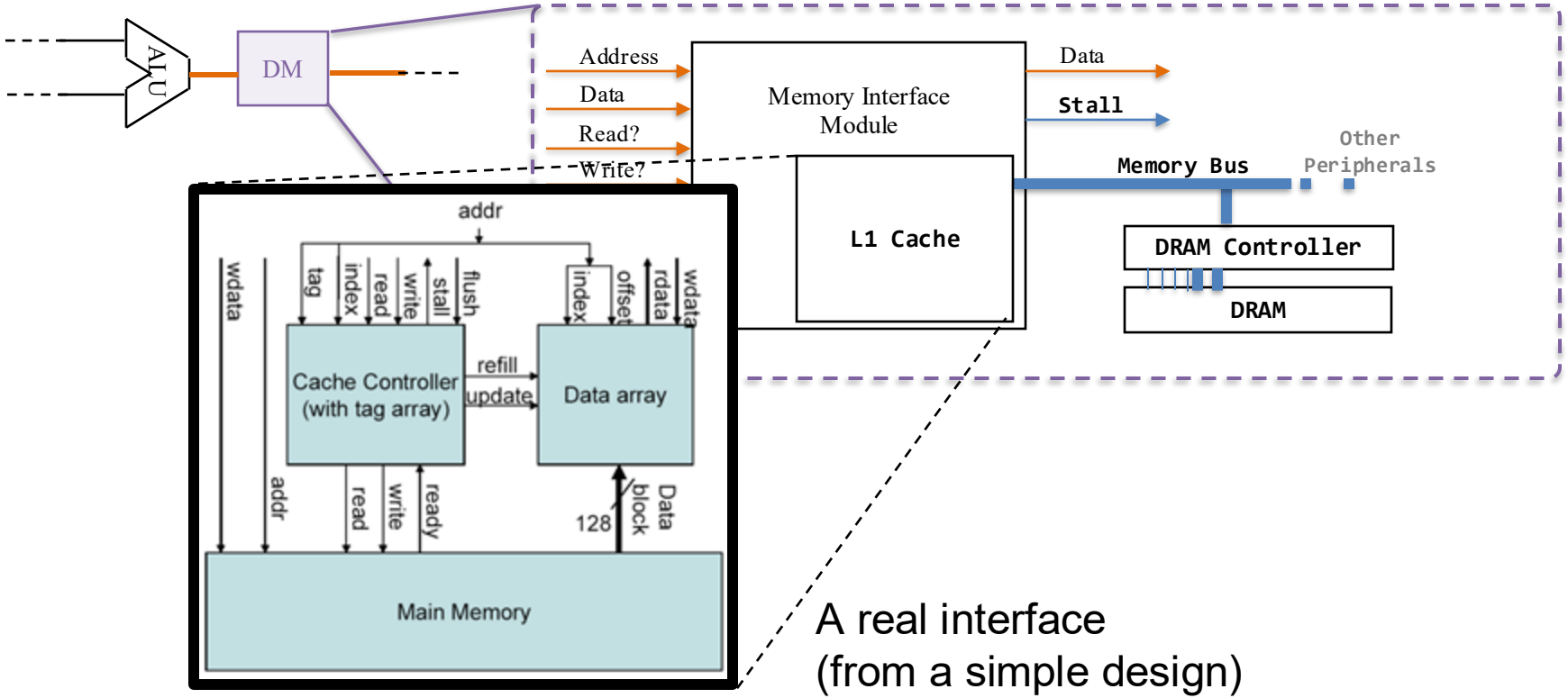
How does the cache actually connect to the pipeline?

[Simplified View! Very different between embedded & high-perf!]



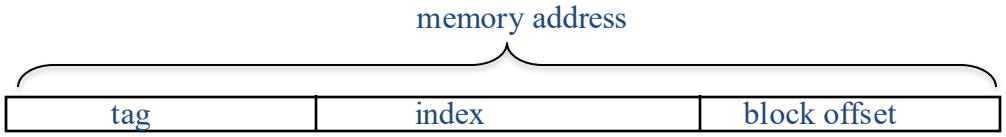
How does the cache actually connect to the pipeline?

[Simplified View! Very different between embedded & high-perf!]

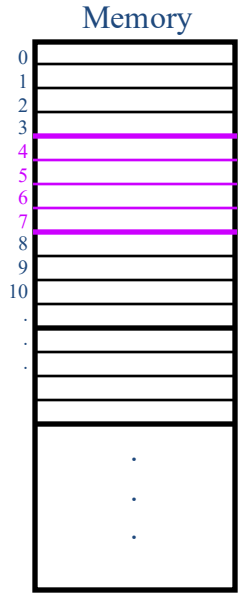


A real interface
(from a simple design)

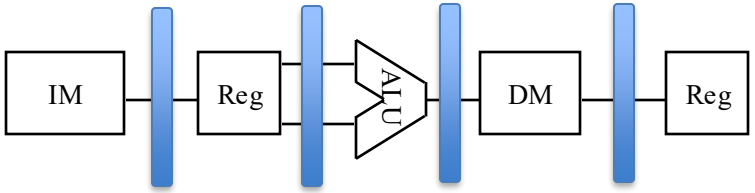
Cache Alignment Revisited



- The data that gets moved into the cache on a miss are all data whose addresses share the same tag and index (regardless of which data gets accessed first).
- Think of main memory as organized into cache-line sized pieces (because in reality, it is!).
- "Block-aligned" access is similar to the "word-aligned" access you have been thinking about already
  - `CPU Core <--> Cache  interface: Word aligned`
  - `Cache <--> Memory interface: Block aligned`



Back to our pipeline, what does all this have to do with performance and alignment?



Which of the following things are possible?

1. It is possible to make a “set-associative” cache to hav N=1 sets?
2. It is possible for a set-associative cache to have N=3 sets?
3. It is possible for a cache to have a 12-byte block size?

Selection	1	2	3
A	Yes	Yes	Yes
B	Yes	Yes	No
C	No	No	Yes
D	No	No	No
E	None of the above		

Associative Caches

- Higher hit rates, but...

- longer access time
  - (longer to determine hit/miss, more muxing of outputs)
- more space (longer tags)
  - 16 KB, 16-byte blocks, DM, tag = ?
  - 16 KB, 16-byte blocks, 4-way, tag = ?

```
for (int i = 0; i < 10,000,000; i++)
      sum += A[i];
```

Assume each element of A is 4 bytes and sum is kept in a register.
Assume a baseline direct-mapped 32KB L1 cache with 32 byte blocks.
Assume this loop is visited many times.
Which changes would help the hit rate of the above code?

Selection	Change
A	Increase to 2-way set associativity
B	Increase block size to 64 bytes
C	Increase cache size to 64 KB
D	A and C combined
E	A, B, and C combined

```
for (int i=0; i < 10,000,000; i++)
      for (int j = 0; j < 8192; j++)
            sum += A[j] – B[j];
```

Assume each element of A and B are 4 bytes.
Assume each array is at least 32KB in size.
Assume sum is kept in a register.
Assume a baseline direct-mapped 32KB L1 cache with 32 byte blocks.

Which changes would help the hit rate of the above code?

Selection	Change
A	Increase to 2-way set associativity
B	Increase block size to 64 bytes
C	Increase cache size to 64 KB
D	A and C combined
E	A, B, and C combined

Dealing with Stores

- Stores must be handled differently than loads, because...
  - they don't necessarily require the CPU to stall.
  - they change the content of cache/memory (creating memory *consistency* issues)
- Q: Can you think of a situation when you might need to load from memory before you can execute a store?
  - Can you think of another one?

Policy decisions for stores

- Keep memory and cache identical?
  - _____ => all writes go to both cache and main memory
  - _____ => writes go only to cache. Modified cache lines are written back to memory when the line is replaced.
- Make room in cache for store miss?
  - *write-allocate* => on a store miss, bring written line into the cache
  - *write-around* => on a store miss, ignore cache

Dealing with stores

- On a store hit, write the new data to cache.
  - In a *write-through* cache, write the data immediately to memory.
  - In a *write-back* cache, mark the line as dirty.
- On a store miss, initiate a cache block load from memory for a write-allocate cache.
  - Write directly to memory for a write-around cache.
- On any kind of cache miss in a write-back cache, if the line to be replaced in the cache is dirty, write it back to memory.

Cache Performance

- CPI = BCPI + MCPI
  - BCPI = base CPI, which means the CPI assuming perfect memory
  - MCPI = the memory CPI, the number of cycles (per instruction) the processor is stalled waiting for memory.

Cache Performance

CPI = BCPI + MCPI

- BCPI = base CPI, which means the CPI assuming perfect memory
- MCPI = the memory CPI, the number of cycles (per instruction) the processor is stalled waiting for memory.

MCPI = accesses/instruction * miss rate * miss penalty

- this assumes we stall the pipeline on both read and write misses, that the miss penalty is the same for both, that cache hits require no stalls.
- If the miss penalty or miss rate is different for Inst cache and data cache (common case), then

```
MCPI = I$ accesses/inst x I$MissRate x I$MissPenalty
     + D$ accesses/inst x D$MissRate x D$MissPenalty
```

In fact…

- Can generalize this “formula” further for other stalls
  - (This is just putting a label on the ”penalty” idea we introduced earlier)

- CPI = BCPI + DHSPI + BHSPI + MCPI
  - DHSPI = data hazard stalls per instruction
  - BHSPI = branch hazard stalls per instruction.

Cache Performance

- Instruction cache miss rate of 4%
- Data cache miss rate of 10%
- BaseCPI = 1.0 (no data or control hazards)
- 20% of instructions are loads and stores
- Miss penalty = 12 cycles

CPI = ???

Selection	CPI (rounded if necessary)
A	1.24
B	1.34
C	1.48
D	1.72
E	None of the above

Cache Performance

- Unified cache
- 25% of instructions are loads and stores
- BaseCPI = 1.2, miss penalty of 10 cycles

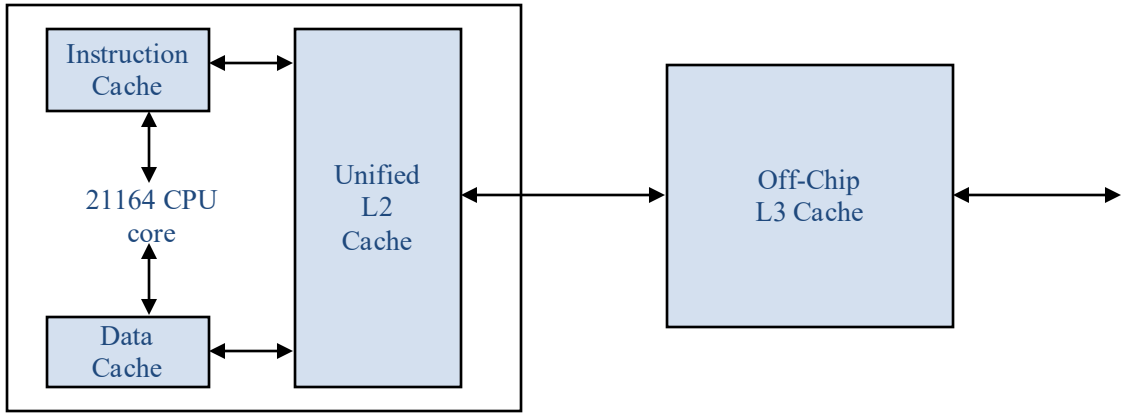
- If we improve the miss rate from 10% to 4% (e.g. with a larger cache), how much do we improve performance?

Cache Performance

- BaseCPI = 1
- Miss rate of 8% overall, 20% loads, miss penalty 20 cycles, never stalls on stores.

- What is the speedup from doubling the CPU clock rate?

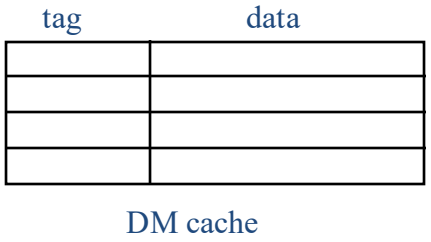
Example -- DEC Alpha 21164 Caches



- ICache and DCache -- 8 KB, DM, 32-byte lines
- L2 cache -- 96 KB, ?-way SA, 32-byte lines
- L3 cache -- 1 MB, DM, 32-byte lines

Three types of cache misses

- Compulsory (or cold-start) misses
  - first access to the data.
- Capacity misses
  - we missed only because the cache isn't big enough.
- Conflict misses
  - we missed because the data maps to the same line as other data that forced it out of the cache.



DM cache

address string:	
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100

Q: Categorizing Misses

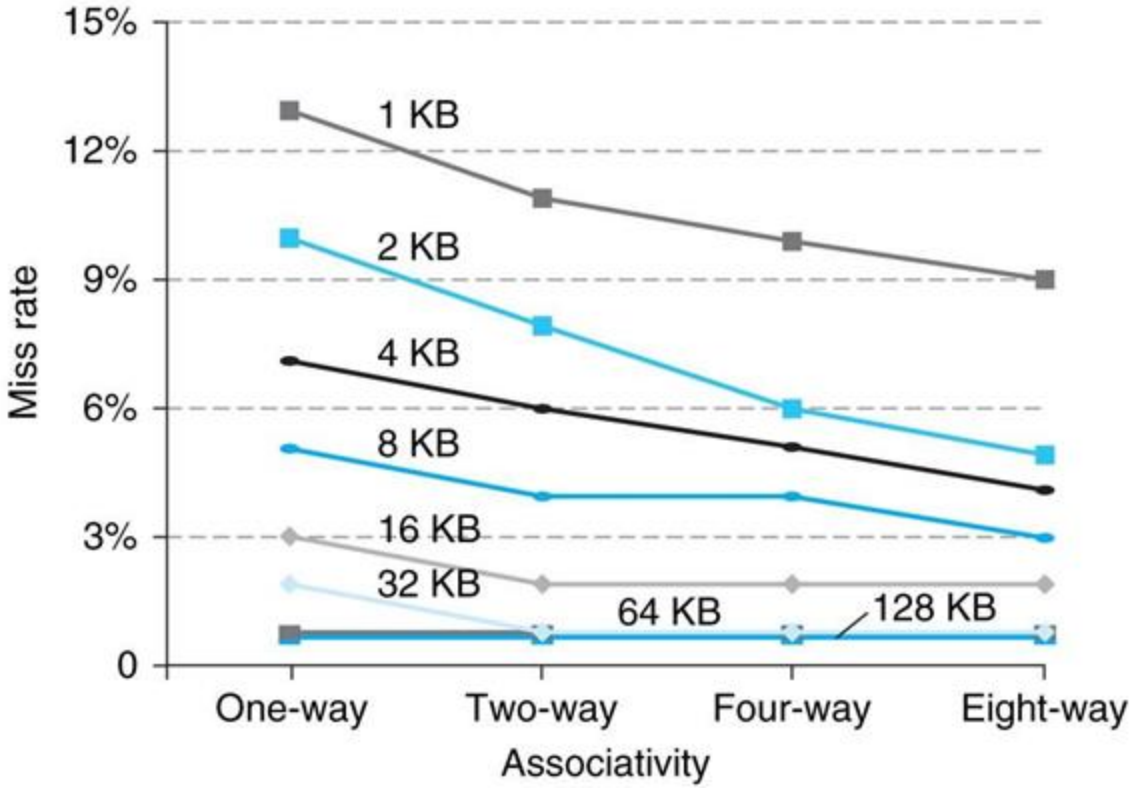
- Suppose you experience a cache miss on a block (let's call it block A).
- You have accessed block A in the past.
- There have been precisely 1027 different blocks accessed between your last access to block A and your current miss.
- Your block size is 32-bytes and you have a 64KB cache. What kind of miss was this?

Selection	Cache Miss
A	Compulsory
B	Capacity
C	Conflict
D	Both Capacity and Conflict
E	None of the above

So, then, how do we decrease...

- Compulsory misses?
- Capacity misses?
- Conflict misses?

Cache Associativity



LRU replacement algorithms

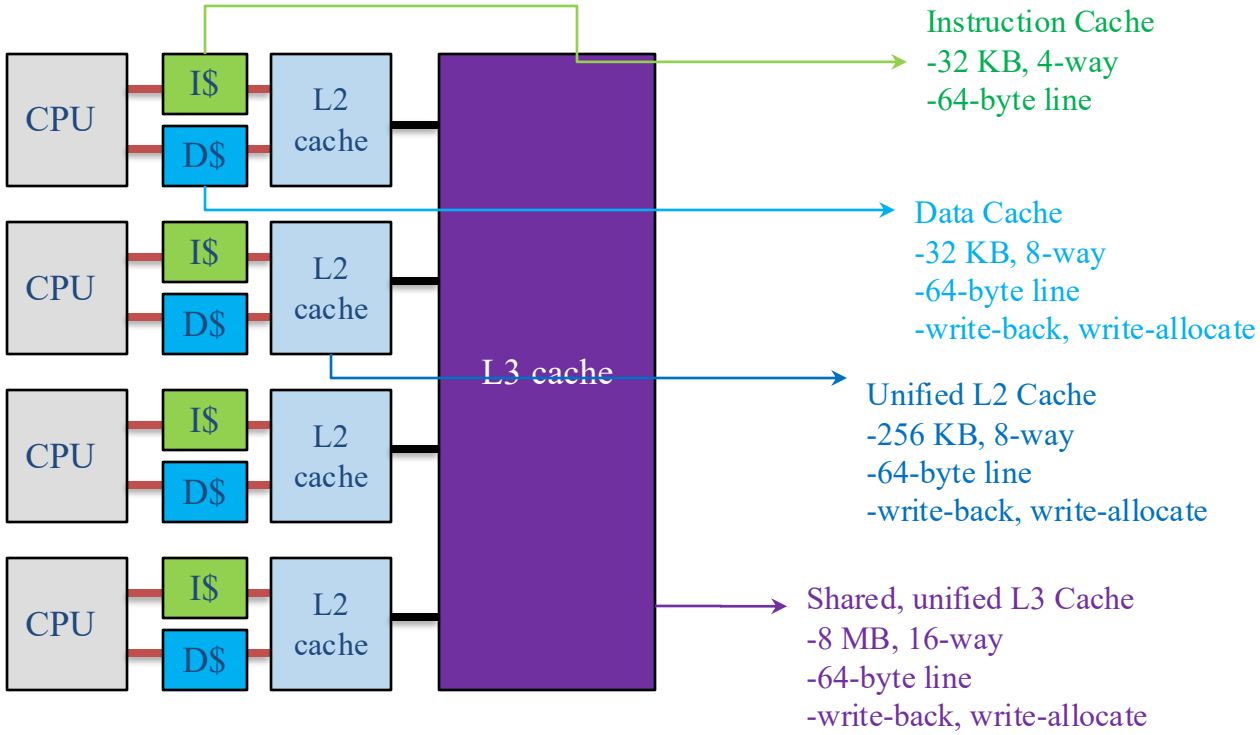
- only needed for associative caches
- requires one bit for 2-way set-associative, 8 bits (per set, 2/line) for 4-way, 24 bits for 8-way…
- can be emulated with log n bits (NMRU)
- can be emulated with use bits for highly associative caches (like page tables)
- However, for most caches (eg, associativity <= 8), LRU is calculated exactly.

Caches in Current Processors

- Not long ago, they were DM at lowest level (closest to CPU), associative further away. Today they are less associative near the processor (2-4+), and more associative farther away (4-16).
- split I and D close (L1) to the processor (for throughput rather than miss rate), unified further away (L2 and beyond).
- write-through and write-back both common, but never write-through all the way to memory.
- 64-byte cache lines common (but getting larger)

- Non-blocking
  - processor doesn't stall on a miss, but only on the use of a miss (if even then)
  - this means the cache must be able to keep track of multiple outstanding accesses, even multiple outstanding misses.

Intel Nehalem (i7)



Key Points

- Caches give illusion of a large, cheap memory with the access time of a fast, expensive memory.
- Caches take advantage of memory locality, specifically **temporal locality** and **spatial locality**.
- Cache design presents many options (block size, cache size, associativity, write policy) that an architect must combine to minimize miss rate and access time to maximize performance.

ADVANCED CACHE ARCHITECTURES

Advanced Cache Architectures

- AMAT = Average Memory Access Time
- AMAT = hit time + miss rate*miss penalty

- (usually expressed in cycles, but can also be expressed in time, e.g. nanoseconds)
- AMAT is a common measure of memory hierarchy performance in architectural studies.

Advanced Cache Architectures

- AMAT = Average Memory Access Time
- AMAT = hit time + miss rate*miss penalty

- So improving memory hierarchy performance means improving AMAT
- In this context, then, there are several ways to improve performance (reduce AMAT):
  - ?
  - ?
  - ?

Advanced Cache Architectures

- AMAT = Average Memory Access Time
- AMAT = hit time + miss rate*miss penalty

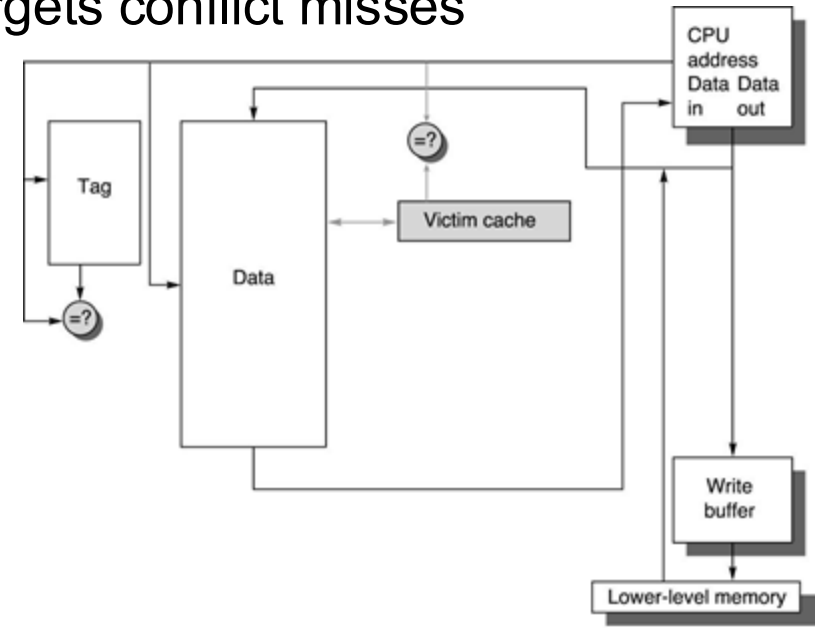
- As a result, then, there are several ways to improve performance (reduce AMAT):
  - Decrease hit time
  - Decrease miss rate
  - Decrease (observed) miss penalty

Hit Rate vs. Hit Time

- Direct-mapped caches have low hit time, associative caches have low miss rate.
- Ideally, we'd like low miss rate and low hit time together.
  - Way prediction
  - Victim Cache

Victim Cache

- Small, fully associative buffer which holds recently evicted cache lines.
- Targets conflict misses



Advanced Cache Architectures

- AMAT = Average Memory Access Time
- AMAT = hit time + miss rate*miss penalty

- As a result, then, there are several ways to improve performance (reduce AMAT):
  - Decrease hit time
  - Decrease miss rate
  - Decrease (observed) miss penalty

Reducing Compulsory Misses and Capacity Misses

- Prefetching
  - Brings data into the cache (or a special buffer) based on access patterns or program knowledge.

Reducing Compulsory Misses and Capacity Misses

- Prefetching
  - Brings data into the cache (or a special buffer) based on access patterns or program knowledge.
- Who does the prefetching?
  - Hardware (based on access patterns)
    - Most modern high-performance processors do this
    - Sometimes called *stream buffers*.
  - Software
    - Most ISAs support some kind of software prefetch
    - Works best for regular computation
  - A separate thread (in a multithreaded processor)
    - We called this *speculative precomputation* (2001)
    - Typically done by distilling a reduced version of the main thread

Advanced Cache Architectures

- AMAT = Average Memory Access Time
- AMAT = hit time + miss rate*miss penalty

- As a result, then, there are several ways to improve performance (reduce AMAT):
  - Decrease hit time
  - Decrease miss rate
  - Decrease (observed) miss penalty

Reducing memory stalls

- A ***non-blocking cache*** is one that can still handle new requests after a miss.
  - Requires some extra bookkeeping to keep everything straight.
- A couple of design options:
  - Hit-under-miss (can have 1 outstanding miss)
    - Can continue to service hits after a miss
    - Stalls on second miss
  - Miss-under-miss
    - Can have up to M outstanding misses at once
- More generally, we want to…

Tolerating cache misses

- Sometimes you can't make the miss go away. But that doesn't mean you have to stall. We *tolerate* misses by continuing to make progress in the face of cache misses.
- Miss tolerance techniques (increasingly effective)
  - Stall on miss (no tolerance)
  -
  -
  -
  -

Tolerating cache misses

- Sometimes you can't make the miss go away. But that doesn't mean you have to stall. We *tolerate* misses by continuing to make progress in the face of cache misses.
- Miss tolerance techniques (increasingly effective)
  - Stall on miss (no tolerance)
  - Stall on use
  -
  -
  -

Tolerating cache misses

- Sometimes you can't make the miss go away. But that doesn't mean you have to stall. We *tolerate* misses by continuing to make progress in the face of cache misses.
- Miss tolerance techniques (increasingly effective)
  - Stall on miss (no tolerance)
  - Stall on use
  - Non-blocking caches
  -
  -

Tolerating cache misses

- Sometimes you can't make the miss go away. But that doesn't mean you have to stall. We *tolerate* misses by continuing to make progress in the face of cache misses.
- Miss tolerance techniques (increasingly effective)
  - Stall on miss (no tolerance)
  - Stall on use
  - Non-blocking caches
  - Out-of-order execution
  -

Tolerating cache misses

- Sometimes you can't make the miss go away. But that doesn't mean you have to stall. We *tolerate* misses by continuing to make progress in the face of cache misses.
- Miss tolerance techniques (increasingly effective)
  - Stall on miss (no tolerance)
  - Stall on use
  - Non-blocking caches
  - Out-of-order execution
  - Multithreaded execution

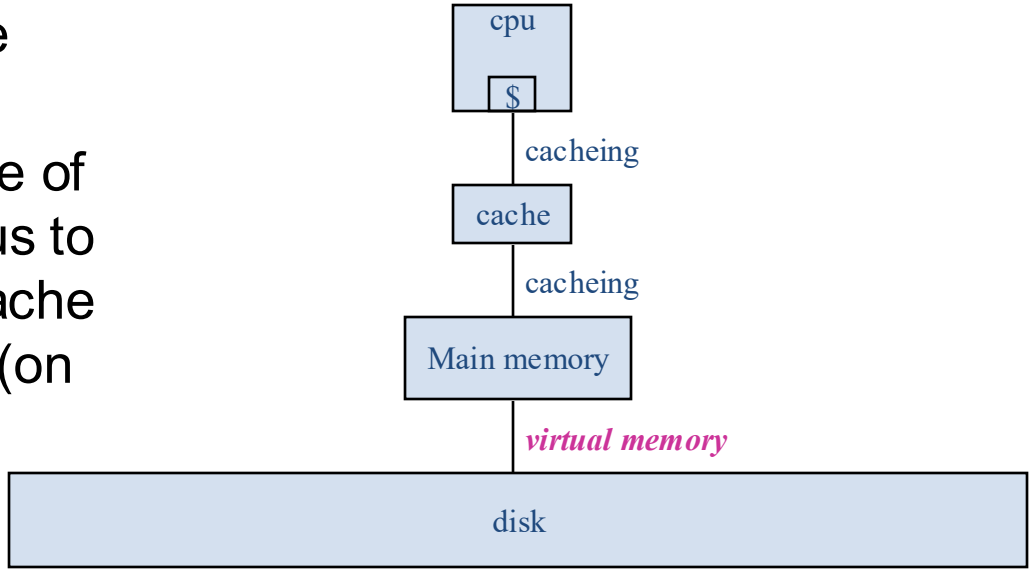
Cache Optimization Summary

- Reducing Conflict Misses
  - Way prediction
  - Victim Cache
- Reducing Capacity or Compulsory Misses
  - Prefefetching
- Tolerating Misses
  - Non-blocking caches
  - Out-of-order execution
  - Multithreading

VIRTUAL MEMORY

Virtual Memory

- It's just another level in the cache/memory hierarchy
- Virtual memory is the name of the technique that allows us to view main memory as a cache of a larger memory space (on disk).



Virtual Memory

- is just cacheing, but uses different terminology (and different storage/lookup techniques)

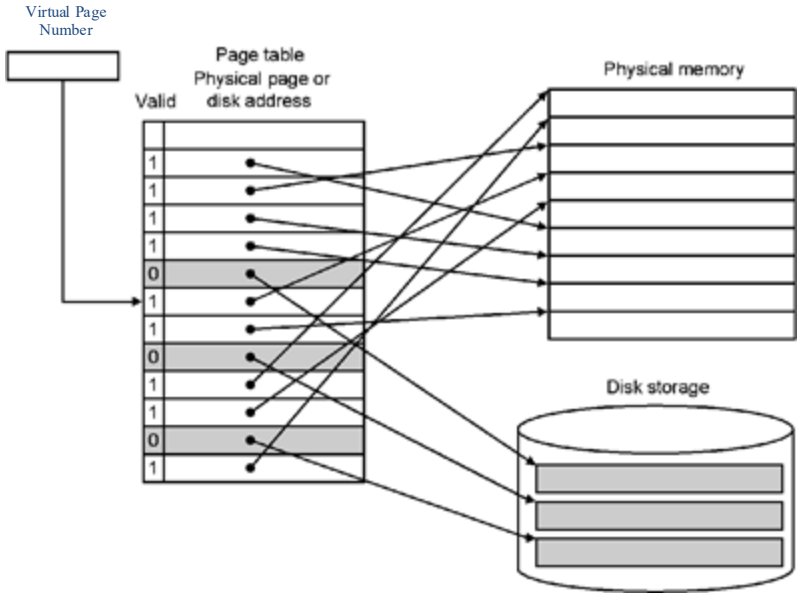
cache	**VM**
block	page
cache miss	page fault
address	virtual address
index	physical address (sort of)

Virtual Memory

- What happens if another program in the processor uses the same addresses that yours does?
- What happens if your program uses addresses that don't exist in the machine?
- What happens to "holes" in the address space your program uses?

- So, virtual memory provides
  - performance (through the cacheing effect)
  - protection
  - ease of programming/compilation
  - efficient use of memory

Virtual Memory…

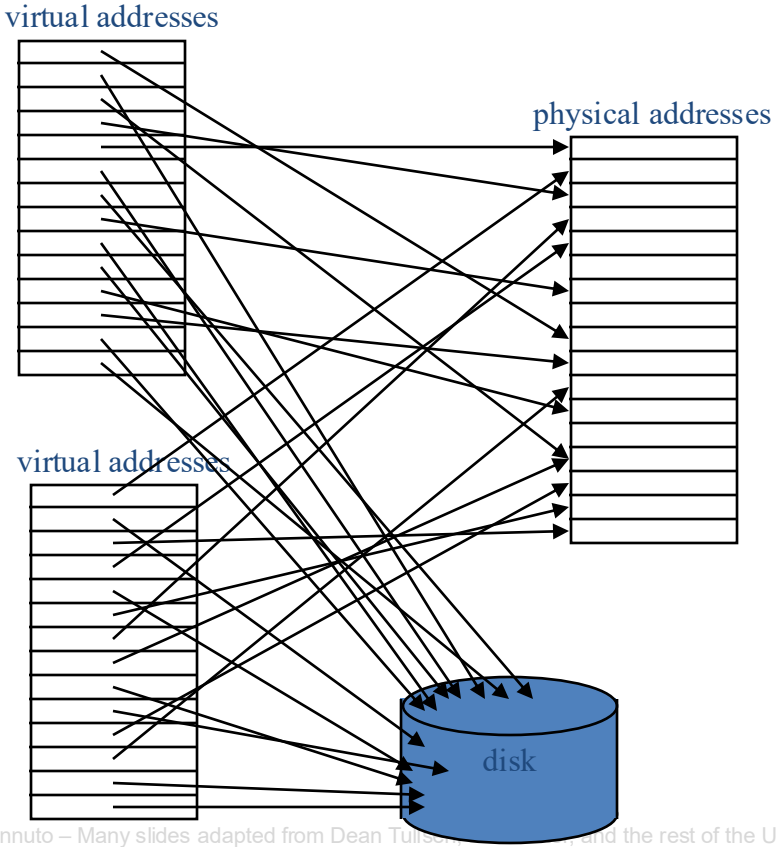


…is just a mapping function from virtual memory addresses to physical memory locations, which allows cacheing of virtual pages in physical memory.

What makes VM different than memory caches

- ***MUCH*** higher miss penalty (millions of cycles)!
- Therefore
  - large pages [equivalent of cache line] (page sizes from 4 KB to MBs)
  - associative mapping of pages (typically fully associative)
  - software handling of misses (but not hits!!)
  - write-through not an option, only write-back

Virtual Memory mapping – Provides process isolation!



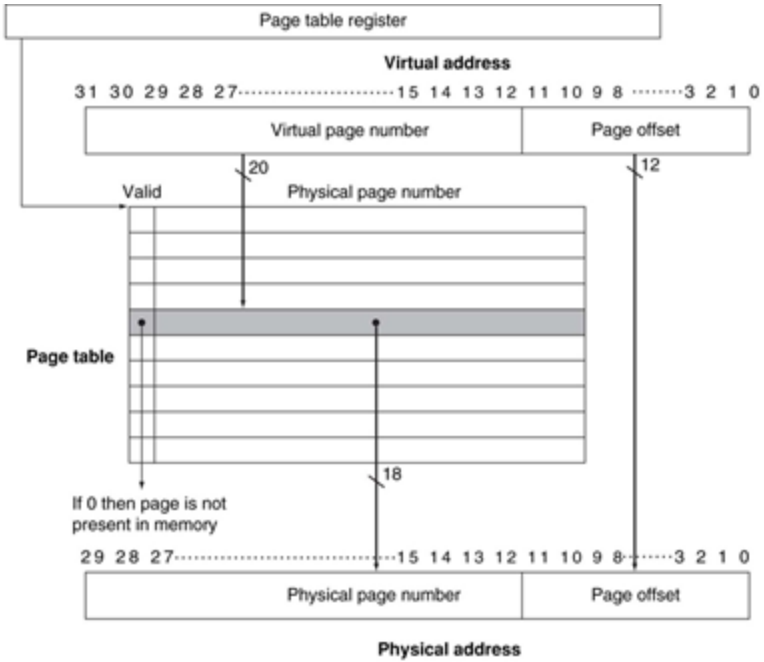
Virtual Address Translation

- We do not need to translate/change all bits of the address.

- We'll only change high order bits, and leave the low order bits alone – the number of low bits we do not change defines the "page size".
  - Page size (virtual memory) is analogous to block size (caches) – it is the chunk of memory that gets moved as a unit on a miss.

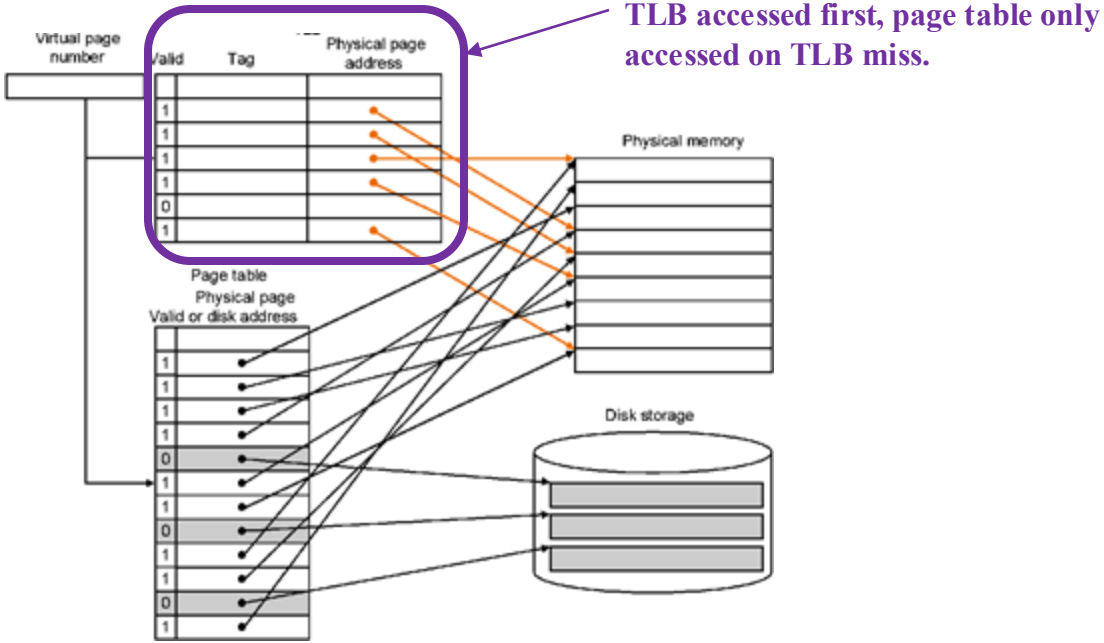
Address translation via the page table

- All page mappings are in the page table, so hit/miss is determined solely by the valid bit
- So why is this fully associative???
  - Where are our “tag” “index” and ”block offset”?
- Biggest problem – this is slow. Why?

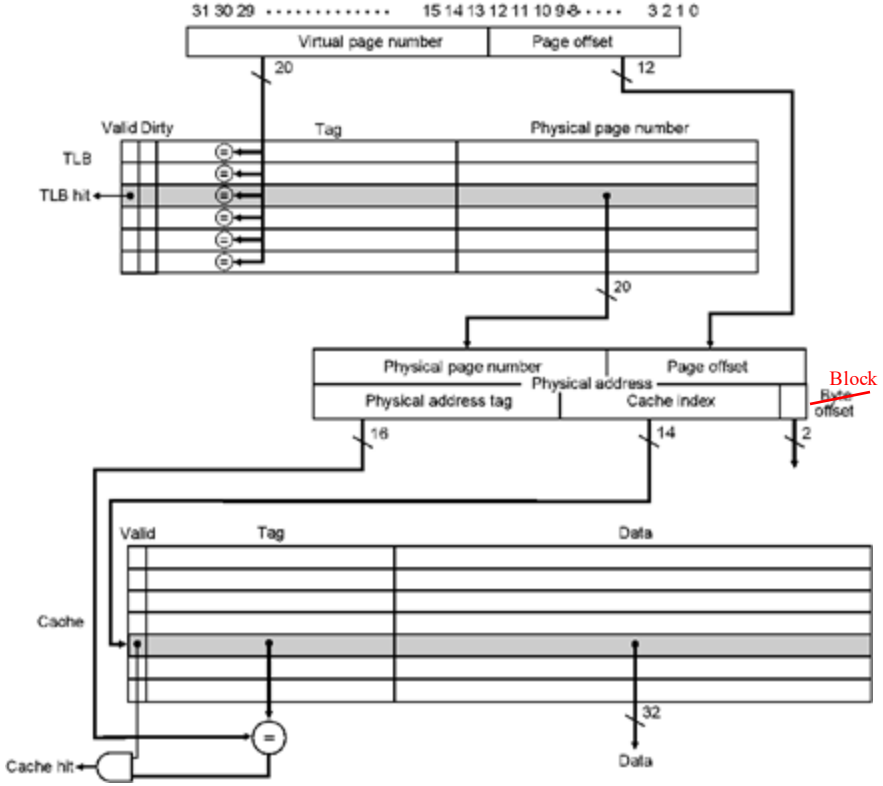


Making Address Translation Fast

- A cache for address translations: translation lookaside buffer (TLB)



TLBs and caches



Virtual Memory & Caches

- Cache lookup is now a serial process
  1. V->P translation through TLB
  2. Get index
  3. Read tag from cache
  4. Compare
- How can we make this faster?
  1.
  2.

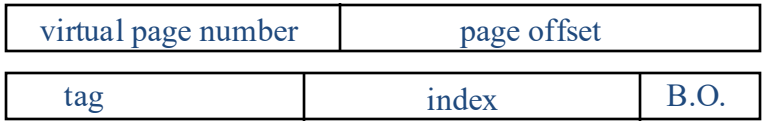
Virtual Caches

- Which addresses are used to lookup data in cache/store in tag?
  - Virtual Addresses?
  - Physical Addresses?
- Pros/Cons?
  - Virtual
  - Physical

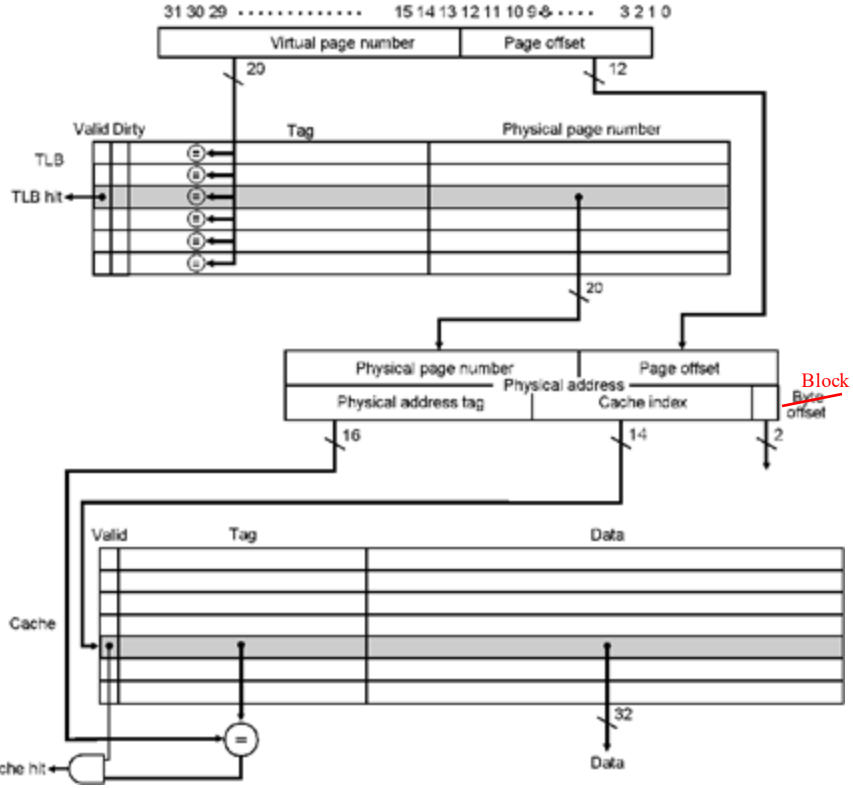
Fast Index Translation

- Can do
  1. V->P translation through TLB
  2. Get index

in parallel, *if* the v->p translation does not change the index.



TLBs and caches



Virtual Memory Key Points

- How does virtual memory provide:
  - protection?
  - sharing?
  - performance?
  - illusion of large main memory?
- Virtual Memory requires twice as many memory accesses, so we cache page table entries in the TLB.
- Three things can go wrong on a memory access: cache miss, TLB miss, page fault.