ECE 4750 Computer Architecture, Fall 2021
T04 Fundamental Memory Microarchitecture

School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-17-17-05

1. Memory Microarchitectural Design Patterns

$$\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left(\frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)$$

Microarchitecture	**Hit Latency**	**Extra Accesses for Translation**
FSM Cache	>1	1+
Pipelined Cache	≈ 1	1+
Pipelined Cache + TLB	≈ 1	≈ 0

1.1. Transactions and Steps

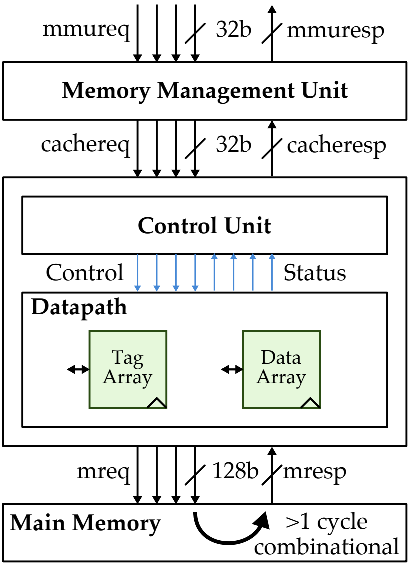
- We can think of each memory access as a transaction
- Executing a memory access involves a sequence of steps

– Check Tag	: Check one or more tags in cache
– Select Victim	: Select victim line from cache using replacement policy
– Evict Victim	: Evict victim line from cache and write victim to memory
– Refill	: Refill requested line by reading line from memory
– Write Mem	: Write requested word to memory
– Access Data	: Read or write requested word in cache

Steps for Write-Through with No Write Allocate

Steps for Write-Back with Write Allocate

1.2. Microarchitecture Overview



2. FSM Cache

$$\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left(\frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)$$

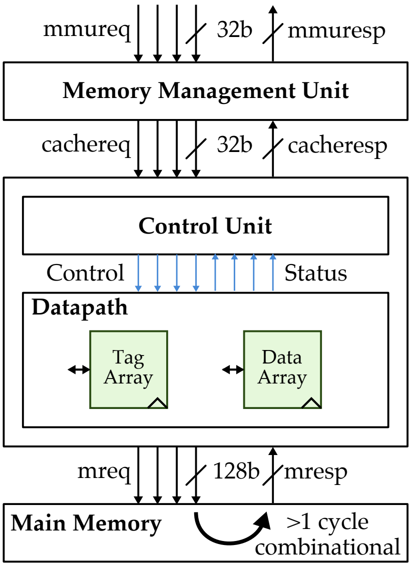
Microarchitecture	Hit Latency	Extra Accesses for Translation
FSM Cache	>1	1+
Pipelined Cache	≈1	1+
Pipelined Cache + TLB	≈1	≈0

Assumptions

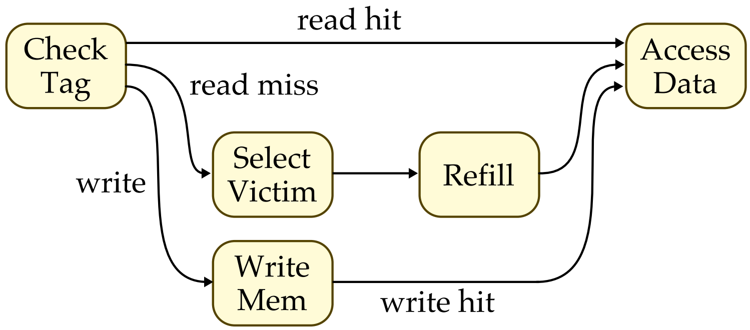
- Page-based translation, no TLB, physically addr cache
- Single-ported combinational SRAMs for tag, data storage
- Unrealistic combinational main memory
- Cache requests are 4 B

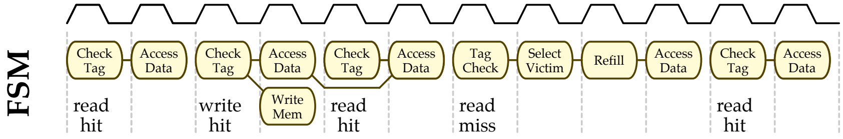
Configuration

- Four 16 B cache lines
- Two-way set-associative
- Replacement policy: LRU
- Write policy: write-through, no write allocate



2.1. High-Level Idea for FSM Cache

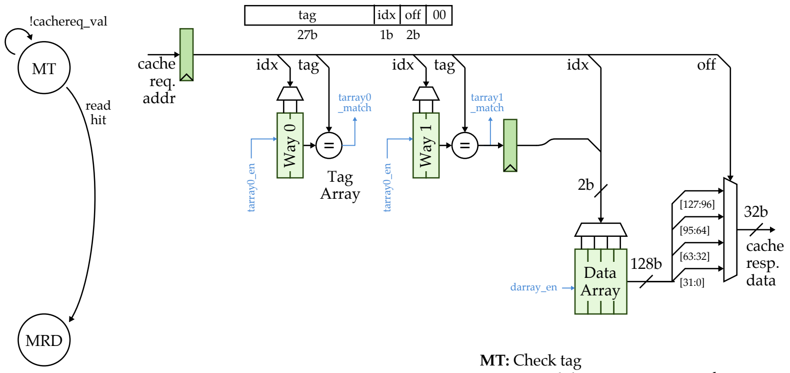




2.2. FSM Cache Datapath

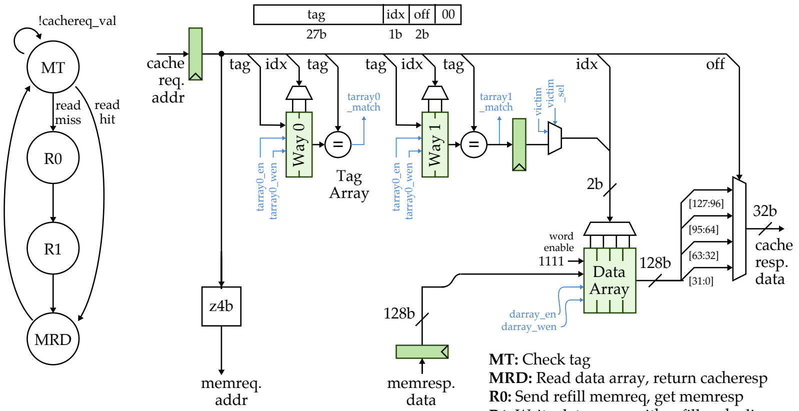
As with processors, we design our cache datapath by incrementally adding support for each transaction and resolving conflicts with muxes.

Implementing READ transactions that hit



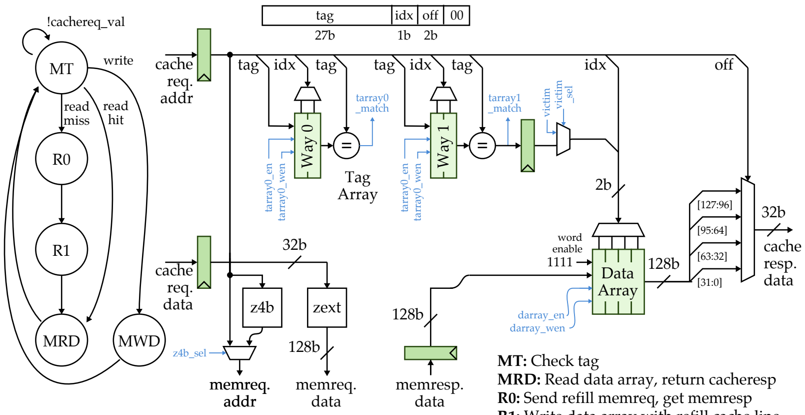
MT: Check tag
MRD: Read data array, return cacheresp

Implementing READ transactions that miss



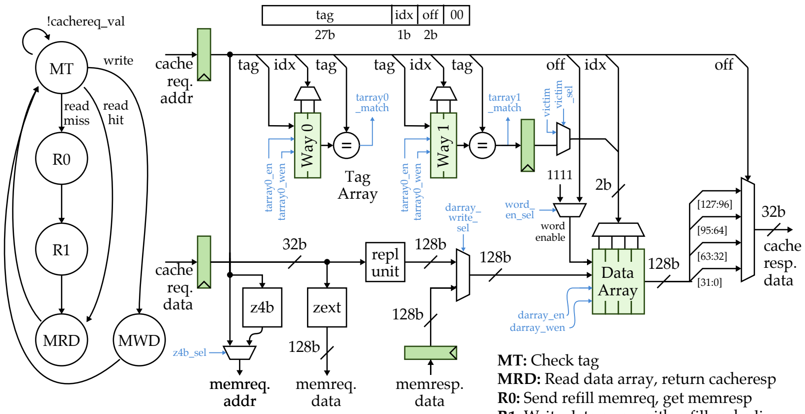
MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line

Implementing WRITE transactions that miss



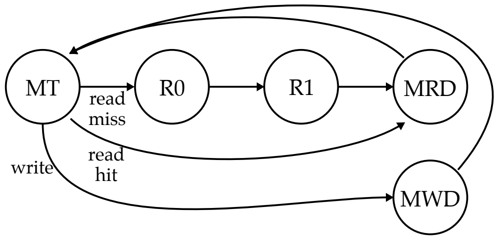
MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array

Implementing WRITE transactions that hit



MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array

2.3. FSM Cache Control Unit



We will need to keep valid bits in the control unit, with one valid bit for every cache line. We will also need to keep use bits which are updated on every access to indicate which was the last "used" line. Assume we create the following two control signals generated by the FSM control unit.

```
hit    =    ( tarray0_match && valid0[idx] )
         || ( tarray1_match && valid1[idx] )

victim = !use[idx]
```

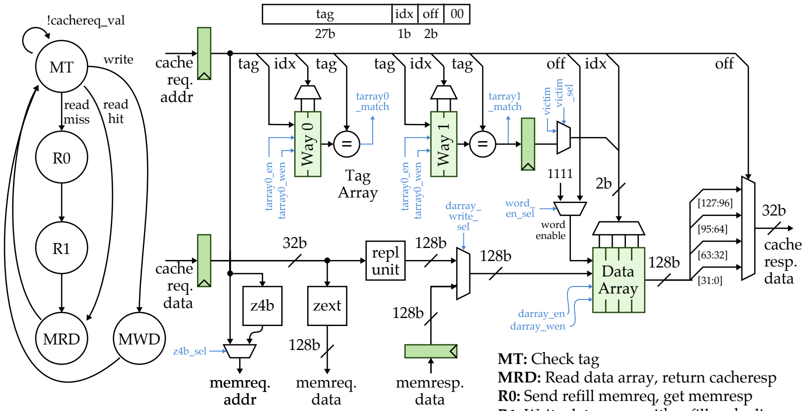
	tarray0		tarray1		darray		darray	worden	z4b	memreq		cachresp
	en	**wen**	**en**	**wen**	**en**	**wen**	**sel**	**sel**	**sel**	**val**	**op**	**val**
MT												
MRD												
R0												
R1												
MWD												

2.4. Analyzing Performance

$$\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left(\frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)$$

Estimating cycle time



MT: Check tag
MRD: Read data array, return cacheresp
R0: Send refill memreq, get memresp
R1: Write data array with refill cache line
MWD: Send write memreq, write data array

- register read/write $= 1\tau$
- tag array read/write $= 10\tau$
- data array read/write $= 10\tau$
- mem read/write $= 20\tau$
- decoder $= 3\tau$
- comparator $= 10\tau$
- mux $= 3\tau$
- repl unit $= 0\tau$
- z4b $= 0\tau$

Estimating AMAL

Consider the following sequence of memory acceses which might correspond to copying 4 B elements from a source array to a destination array. Each array contains 64 elements. What is the AMAL?

```
rd 0x1000
wr 0x2000
rd 0x1004
wr 0x2004
rd 0x1008
wr 0x2008
...
rd 0x1040
wr 0x2040
```

Consider the following sequence of memory acceses which might correspond to incrementing 4 B elements in an array. The array contains 64 elements. What is the AMAL?

```
rd 0x1000
wr 0x1000
rd 0x1004
wr 0x1004
rd 0x1008
wr 0x1008
...
rd 0x1040
wr 0x1040
```

3. Pipelined Cache

$$\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left(\frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)$$

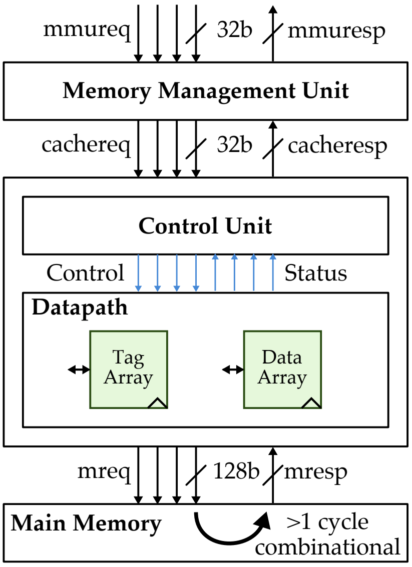
Microarchitecture	Hit Latency	Extra Accesses for Translation
FSM Cache	>1	1+
Pipelined Cache	≈ 1	1+
Pipelined Cache + TLB	≈ 1	≈ 0

Assumptions

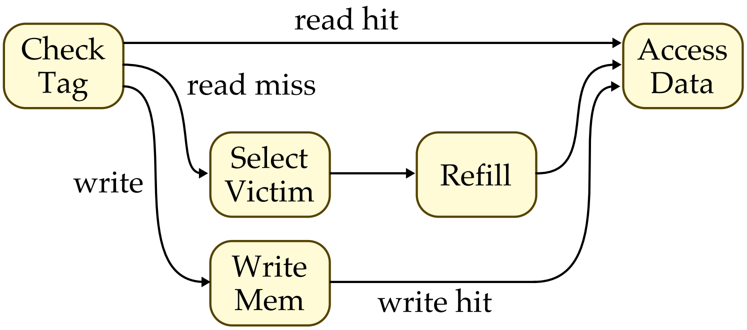
- Page-based translation, no TLB, physically addr cache
- Single-ported combinational SRAMs for tag, data storage
- Unrealistic combinational main memory
- Cache requests are 4 B

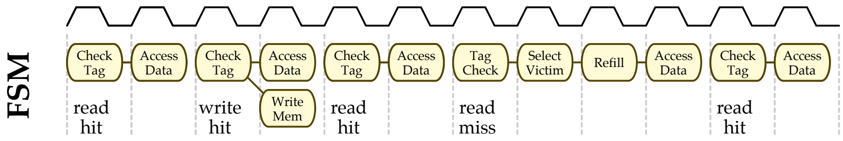
Configuration

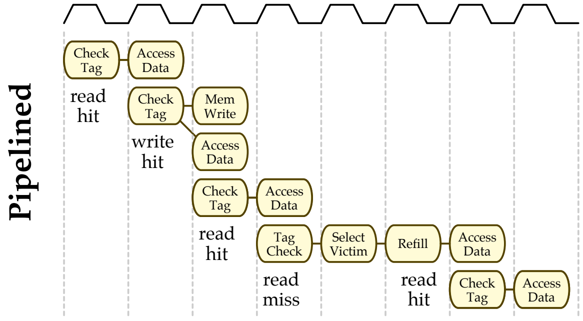
- Four 16 B cache lines
- Direct-mapped
- Replacement policy: LRU
- Write policy: write-through, no write allocate



3.1. High-Level Idea for Pipelined Cache



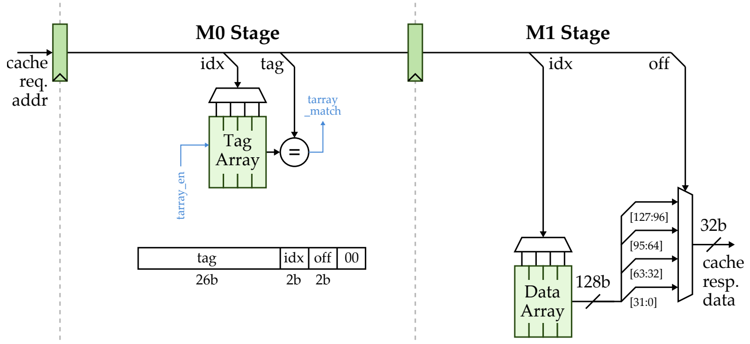




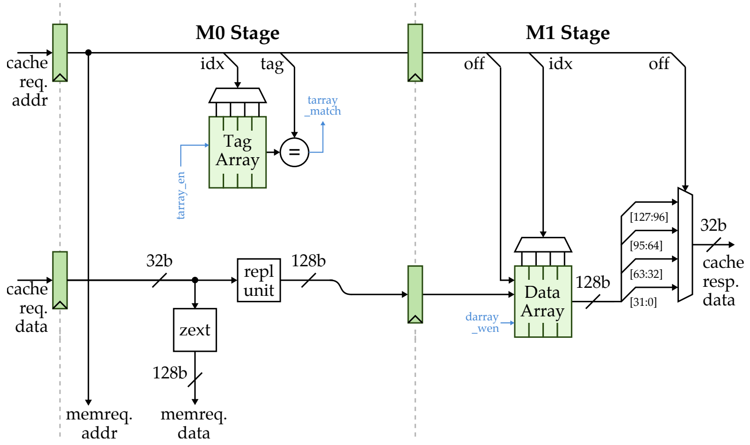
3.2. Pipelined Cache Datapath and Control Unit

As with processors, we incrementally adding support for each transaction and resolving conflicts using muxes.

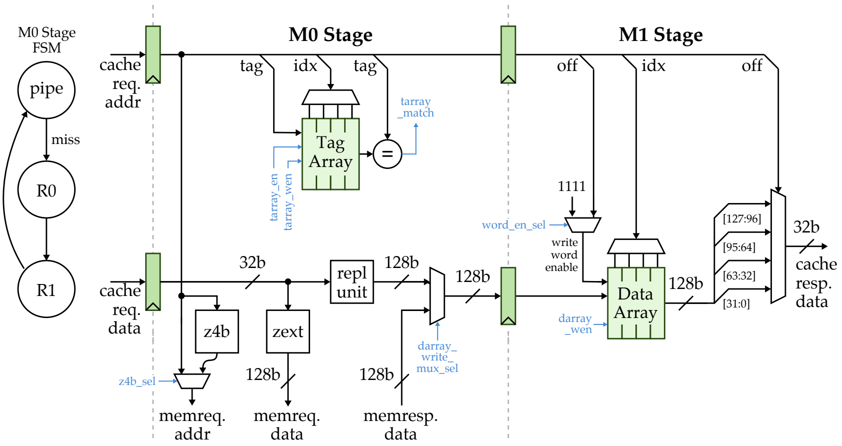
Implementing READ transactions that hit



Implementing WRITE transactions that hit



Implementing transactions that miss

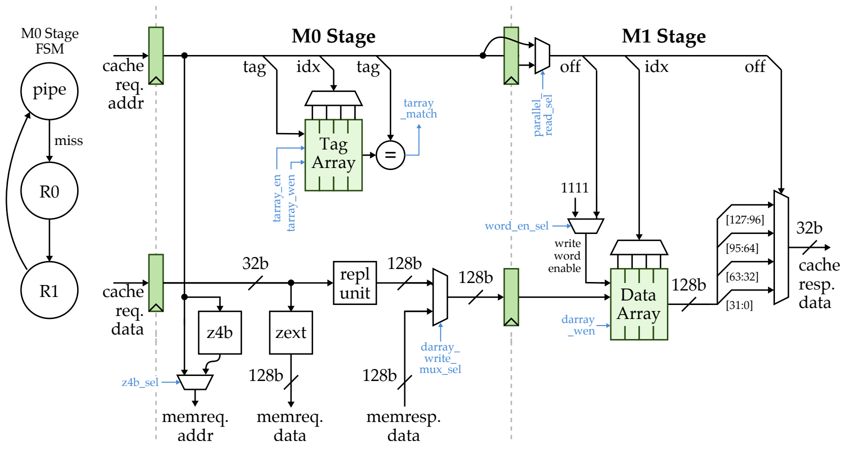


- Hybrid pipeline/FSM design pattern
- Hit path is pipelined with two-cycle hit latency
- Miss path stalls in M0 stage to refill cache line

Pipeline diagram for pipelined cache with 2-cycle hit latency

`rd (hit)`													
`wr (hit)`													
`rd (miss)`													
`rd (miss)`													
`wr (miss)`													
`rd (hit)`													

Parallel read with pipelined write datapath



Pipeline diagram for parallel read with pipelined write

`rd (hit)`														
`rd (hit)`														
`rd (hit)`														
`wr (hit)`														
`wr (hit)`														
`rd (hit)`														

- Achieves single-cycle hit latency for reads
- Two-cycle hit latency for writes, but is this latency observable?
- With write acks, send write-ack back in M0 stage
- How do we resolve structural hazards?

Resolving structural hazard by exposing in ISA

`rd (hit)`														
`wr (hit)`														
`nop`														
`rd (hit)`														

Resolving structural hazard with hardware stalling

`rd (hit)`														
`wr (hit)`														
`rd (hit)`														

```
ostall_M0 = val_M0 && ( type_M0 == RD )
         && val_M1 && ( type_M1 == WR )
```

Resolving structural hazard with hardware duplication

`rd (hit)`														
`wr (hit)`														
`rd (hit)`														

Resolving RAW data hazard with software scheduling

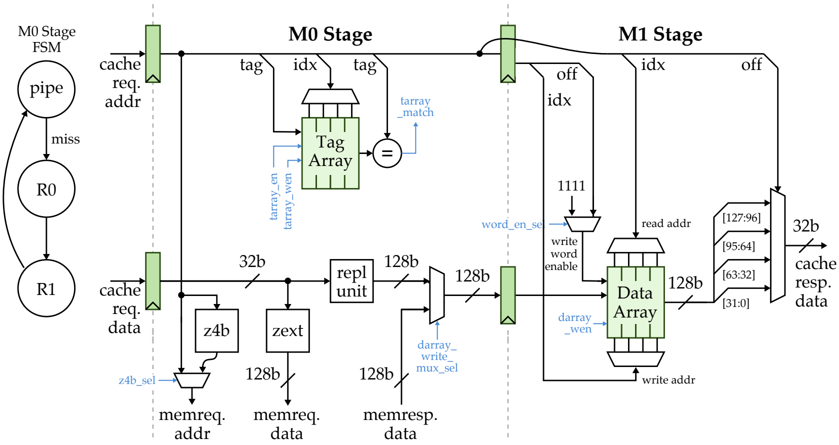
Software scheduling: hazard depends on memory address, so difficult to know at compile time!

Resolving RAW data hazard with hardware stalling

```
ostall_M0 = ...
```

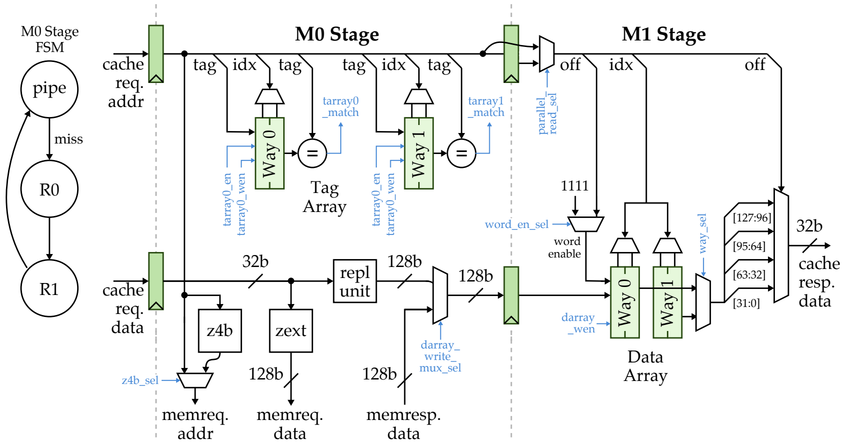
Resolving RAW data hazard with hardware bypassing

We could use the previous stall signal as our bypass signal, but we will also need a new bypass path in our datapath. Draw this new bypass path on the following datapath diagram.



Parallel read and pipelined write in set associative caches

To implement parallel read in set-associative caches, we must speculatively read a line from each way in parallel with tag check. This can be expensive in terms of latency, motivating two-cycle hit latencies for highly associative caches.

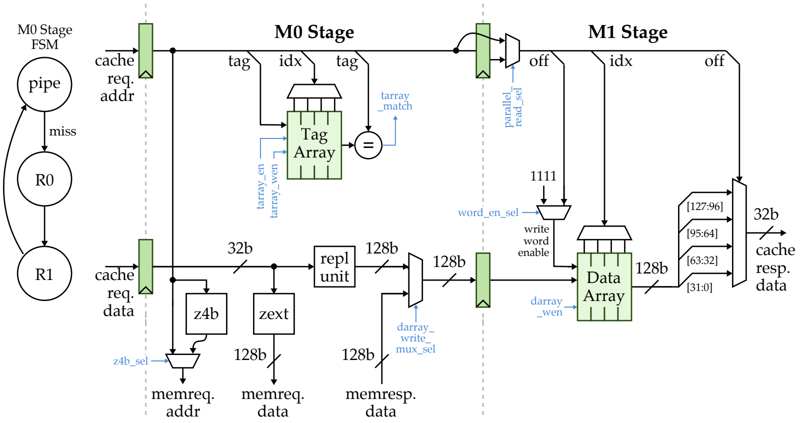


3.3. Analyzing Performance

$$\frac{\text{Time}}{\text{Mem Access Sequence}} = \frac{\text{Mem Accesses}}{\text{Sequence}} \times \frac{\text{Avg Cycles}}{\text{Mem Access}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\frac{\text{Avg Cycles}}{\text{Mem Access}} = \frac{\text{Avg Cycles}}{\text{Hit}} + \left(\frac{\text{Num Misses}}{\text{Num Accesses}} \times \frac{\text{Avg Extra Cycles}}{\text{Miss}} \right)$$

Estimating cycle time



• register read/write	= 1τ
• tag array read/write	= 10τ
• data array read/write	= 10τ
• mem read/write	= 20τ
• decoder	= 3τ
• comparator	= 10τ
• mux	= 3τ
• repl unit	= 0τ
• z4b	= 0τ

Estimating AMAL

Assume a parallel-read/pipelined-write microarchitecture with hardware duplication to resolve the structural hazard and stalling to resolve RAW hazards. Consider the following sequence of memory accesses which might correspond to copying 4 B elements from a source array to a destination array. Each array contains 64 elements. What is the AMAL?

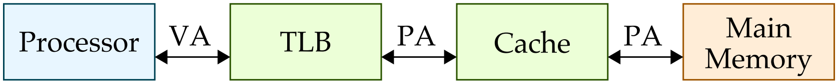
`rd 0x1000`																				
`wr 0x2000`																				
`rd 0x1004`																				
`wr 0x2004`																				
`rd 0x1008`																				
`wr 0x2008`																				
`rd 0x100c`																				
`wr 0x200c`																				
`wr 0x1010`																				

3.4. Pipelined Cache with TLB

How should we integrate a MMU (TLB) into a pipelined cache?

Physically Addressed Caches

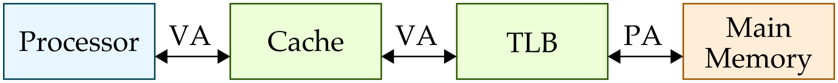
Perform memory translation before cache access

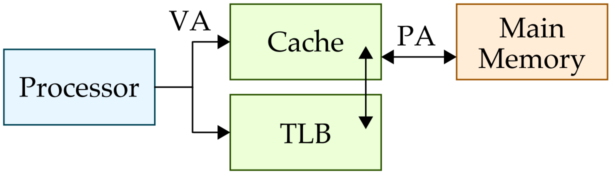


- Advantages:
  - Physical addresses are unique, so cache entries are unique
  - Updating memory translation simply requires changing TLB
- Disadvantages:
  - Increases hit latency

Virtually Addressed Caches

Perform memory translation after or in parallel with cache access

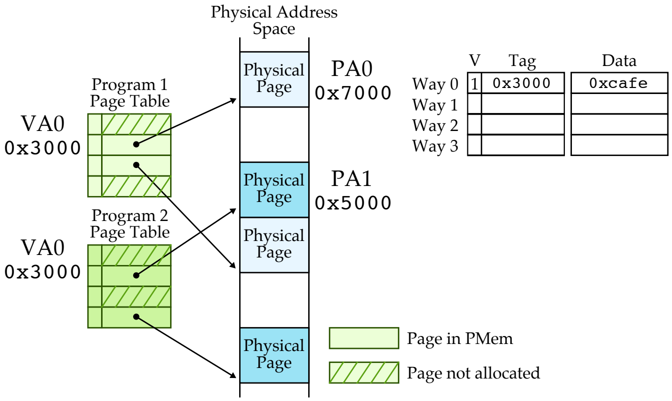




- Advantages:
  - Simple one-step process for hits
- Disadvantages:
  - Intra-program protection (store protection bits in cache?)
  - I/O uses physical addr (map into virtual addr space?)
  - Virtual address homonyms
  - Virtual address synonyms (aliases)

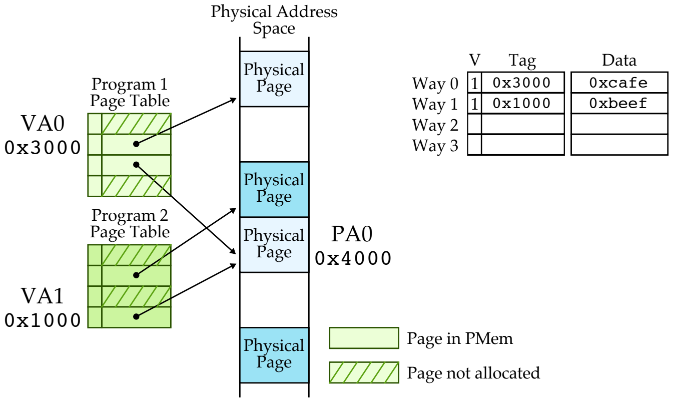
Virtual Address Homonyms

Single virtual address points to two physical address.



- Example scenario
  - Program 1 brings VA 0x3000 into cache
  - Program 1 is context swapped for program 2
  - Program 2 hits in cache, but gets incorrect data!
- Potential solutions
  - Flush cache on context swap
  - Store program ids (address space IDS) in cache

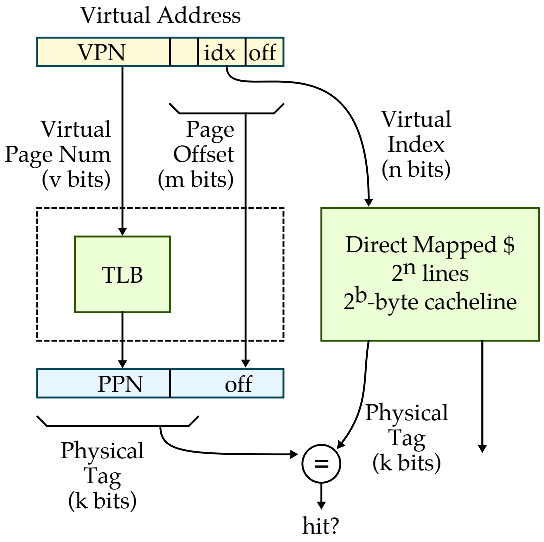
Virtual Address Synonyms (Aliases)



	V	Tag	Data
Way 0	1	0x3000	0xcafe
Way 1	1	0x1000	0xbeef
Way 2			
Way 3			

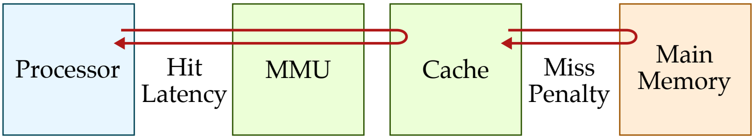
- Example scenarios
  - User and OS can potentially have different VAs point to same PA
  - Memory map the same file (via `mmap`) in two different programs
- Potential solutions
  - Hardware checks all ways (and potentially different sets) on a miss to ensure that a given physical address can only live in one location in cache
  - Software forces aliases to share some address bits (page coloring) reducing the number of sets we need to check on a miss (or reducing the need to check any other locations for a direct mapped cache)

Virtually Indexed and Physically Tagged Caches



- Page offset is the same in VA and PA
- Up to n bits of physical address available without translation
- If index bits + cache offset (n+b) < page offset bits (m), can do translation in parallel with reading out the physical tag and data
- Complete (physical) tag check once tag/data access complete
- With 4 KB pages, direct-mapped cache must be <= 4 KB
- Larger page sizes (decrease k) or higher associativity (decrease n) enable larger virtually indexed, physically tagged caches

4. Cache Microarchitecture Optimizations

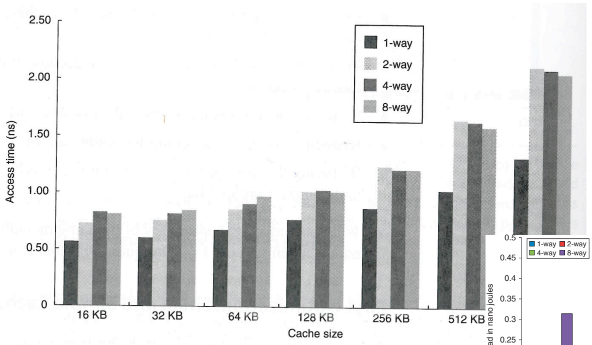


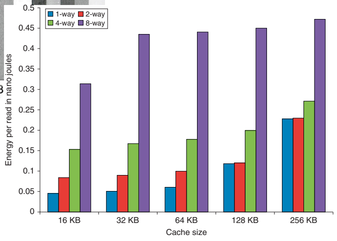
$$\text{AMAL} = \text{Hit Latency} + (\text{ Miss Rate} \times \text{Miss Penalty })$$

- Reduce hit time
  - Small and simple caches
- Reduce miss penalty
  - Multi-level cache hierarchy
  - Prioritize reads
- Reduce miss rate
  - Large block size
  - Large cache size
  - High associativity
  - Hardware prefetching
  - Compiler optimizations

4.1. Reduce Hit Time

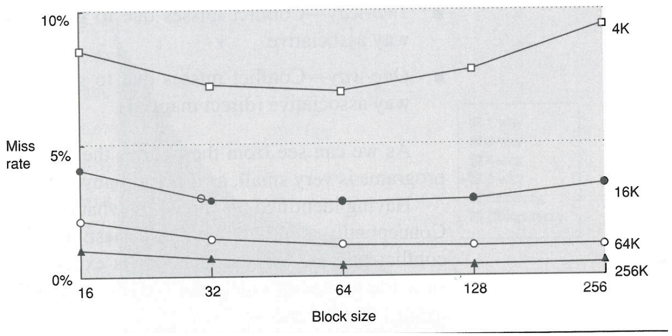
Cache Microarchitecture Optimizations





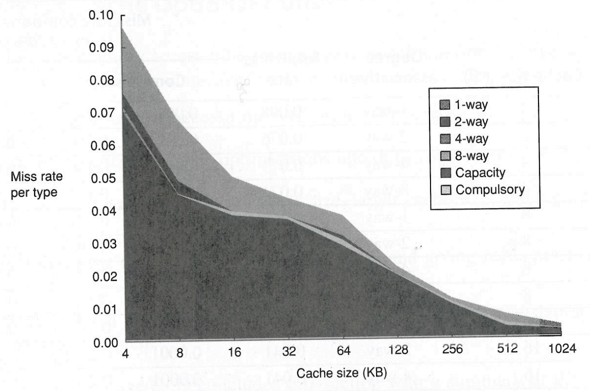
4.2. Reduce Miss Rate

Large Block Size



- Less tag overhead
- Exploit fast burst transfers from DRAM and over wide on-chip busses
- Can waste bandwidth if data is not used
- Fewer blocks → more conflicts

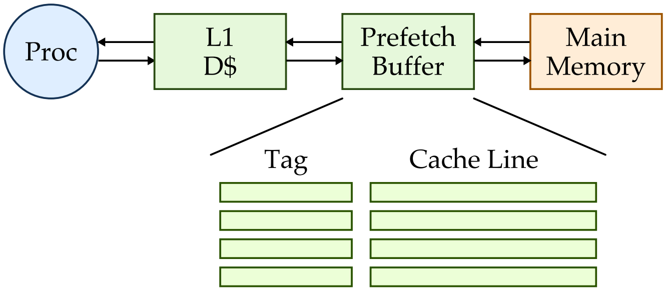
Large Cache Size or High Associativity



If cache size is doubled, miss rate usually drops by about $\sqrt{2}$

Direct-mapped cache of size N has about the same miss rate as a two-way set-associative cache of size N/2

Hardware Prefetching



- Previous techniques only help capacity and conflict misses
- Hardware prefetcher looks for patterns in miss address stream
- Attempts to predict what the next miss might be
- Prefetches this next miss into a pfetch buffer
- Very effective in reducing compulsory misses for streaming accesses

Compiler Optimizations

- Restructuring code affects the data block access sequence
  - Group data accesses together to improve spatial locality
  - Re-order data accesses to improve temporal locality
- Prevent data from entering the cache
  - Useful for variales that will only be accessed once before eviction
  - Needs mechanism for software to tell hardware not to cache data ("no-allocate" instruction hits or page table bits)
- Kill data that will never be used again
  - Streaming data exploits spatial locality but not temporal locality
  - Replace into dead-cache locations

Loop Interchange and Fusion

What type of locality does each optimization improve?

```
for(j=0; j < N; j++) {
     for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
     }
 }
```

↓

```
 for(i=0; i < M; i++) {
     for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
     }
 }
```

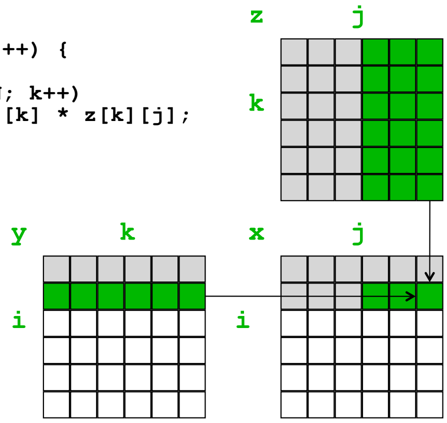
```
for(i=0; i < N; i++)
      a[i] = b[i] * c[i];

for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
```

↓

```
for(i=0; i < N; i++)
{
       a[i] = b[i] * c[i];
       d[i] = a[i] * c[i];
}
```

Matrix Multiply with Naive Code

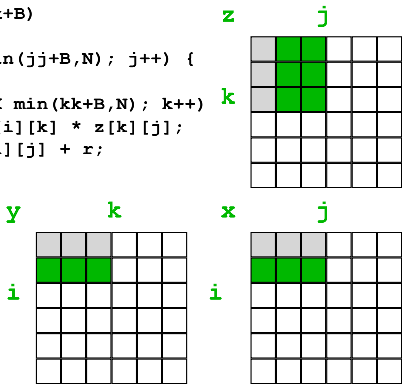
```
for(i=0; i < N; i++)
    for(j=0; j < N; j++) {
       r = 0;
       for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
       x[i][j] = r;
    }
```



☐ *Not touched* ▨ *Old access* 🟩 *New access*

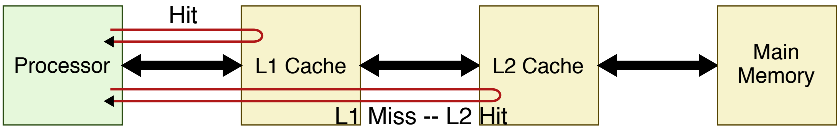
Matrix Multiply with Cache Tiling

```
for(jj=0; jj < N; jj=jj+B)
     for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
               r = 0;
               for(k=kk; k < min(kk+B,N); k++)
                  r = r + y[i][k] * z[k][j];
               x[i][j] = x[i][j] + r;
            }
```



4.3. Reduce Miss Penalty

Multi-Level Caches



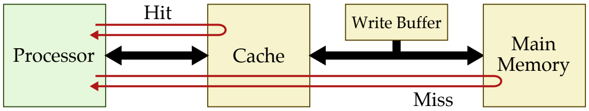
$$\text{AMAL}_{L1} = \text{Hit Latency of L1} + (\text{ Miss Rate of L1} \times \text{AMAL}_{L2}\text{ })$$

$$\text{AMAL}_{L2} = \text{Hit Latency of L2} + (\text{ Miss Rate of L2} \times \text{Miss Penalty of L2 })$$

- Local miss rate = misses in cache / accesses to cache
- Global miss rate = misses in cache / processor memory accesses
- Misses per instruction = misses in cache / number of instructions

- Use smaller L1 is there is also a L2
  - Trade increased L1 miss rate for reduced L1 hit time & L1 miss penalty
  - Reduces average access energy
- Use simpler write-through L1 with on-chip L2
  - Write-back L2 cahce absorbs write traffic, doesn't go off-chip
  - Simplifies processor pipeline
  - Simplifies on-chip coherence issues
- Inclusive Multilevel Cache
  - Inner cache holds copy of data in outer cache
  - External coherence is simpler
- Exclusive Multilevel Cache
  - Inner cache may hold data in outer cache
  - Swap lines between inner/outer cache on miss

Prioritize Reads



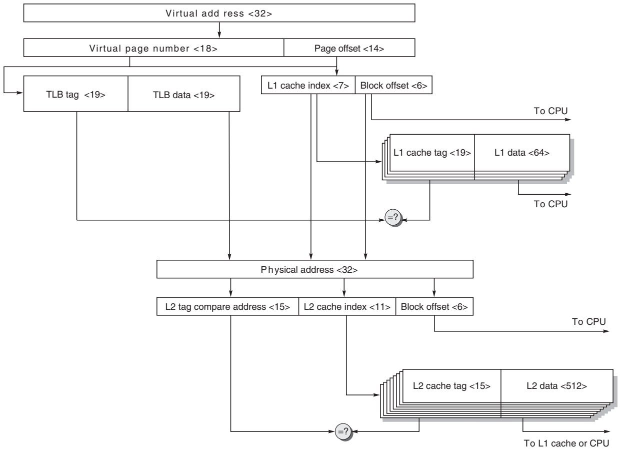
- Processor not stalled on writes, and read misses can go ahead of writes to main memory
- Write buffer may hold updated value of location needed by read miss
  - On read miss, wait for write buffer to be empty
  - Check write buffer addresses and bypass

4.4. Cache Optimization Summary

Technique	Hit Lat	Miss Rate	Miss Penalty	BW	HW
Smaller caches	–	+			0
Avoid TLB before indexing	–				1
Large block size		–	+		0
Large cache size	+	–			1
High associativity	+	–			1
Hardware prefetching		–			2
Compiler optimizations		–			0
Multi-level cache			–		2
Prioritize reads			–		1
Pipelining	+			+	1

5. Case Study: ARM Cortex A8 and Intel Core i7

5.1. ARM Cortex A8



- L1 data cache
  - 32 KB, 64 B cache lines, 4-way set-associative with random replacement
  - 32K/64 = 512 cache lines, 128 lines per set (set index = 7 bits)
  - Virtually indexed, physically tagged with single-cycle hit latency
- Memory management unit
  - TLB with multi-level page tables in physical memory
  - TLB has 32 entries, fully associative, hardware TLB miss handler
  - Variable page size: 4 KB, 16 KB, 64 KB, 1 MB, 16 MB (fig shows 16 KB)
  - TLB tag entries can have wildcards to support multiple page sizes
- L2 cache
  - 1 MB, 64 B cache lines, 8-way set-associative
  - 1M/64 = 16K cache lines, 2K lines per set (set index = 11 bits)
  - Physically addressed with multi-cycle hit latency

5.2. Intel Core i7

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

- Write-back with merging write buffer (more like no write allocate)
- L3 is inclusive of L1/L2
- Hardware prefetching from L2 into L1, from L3 into L2
- Virtually indexed, physically tagged L1 caches
- Physically addressed L2/L3 caches

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Size	128	64	512
Associativity	4-way	4-way	4-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	6 cycles
Miss	7 cycles	7 cycles	Hundreds of cycles to access page table

- 48 bit virtual addresses and 36 bit physical addresses (36 GB physical mem)
- 4 KB pages except for few large 2–4MB pages in L1 TLBs