16. Advanced pipelining and memory system introduction

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor

© Austin & Papaefthymiou, 2009
The material in this presentation cannot be copied in any form without our written permission

Announcements

- Homework 4 due today.
- Homework 5 going out today.
- Homework 2 has been graded (available outside my office starting later today)

Last Pipelining Hurrah: Out of Order Execution

- Eliminating stall conditions decreases CPI
- Reorder instructions to avoid stalls
- Example (5-stage LC2K pipeline):

add 1 2 3	add 1 2 3
lw 3 2 16	lw 3 2 16
nand 2 6 7	add 4 5 1
add 4 5 1	nand 2 6 7

© Austin & Papaefthymiou – 2009

Why Use Out of Order Execution?

- Some instructions take a long time to execute
  - Floating point operations
  - Some loads and stores (more when we talk about memory hierarchy)
- Options:
  - Increase cycle time
  - Increase number of pipeline stages
  - Execute other instructions while you wait

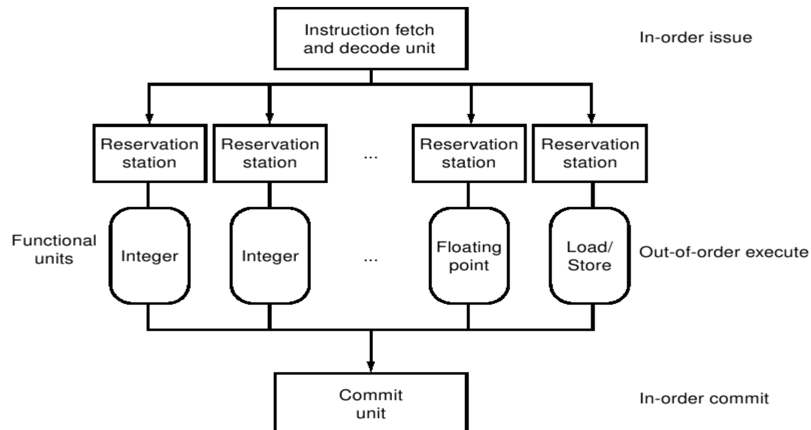
© Austin & Papaefthymiou – 2009

Implementing Out of Order Execution

- Compiler can schedule instructions to avoid stalls
  - needs knowledge of pipeline
  - does not violate compatibility, only affects performance
- Alternatively, the hardware can reorder instructions
  - views pipeline dynamically
  - same instructions can be reordered differently

© Austin & Papaefthymiou – 2009

Out of order execution



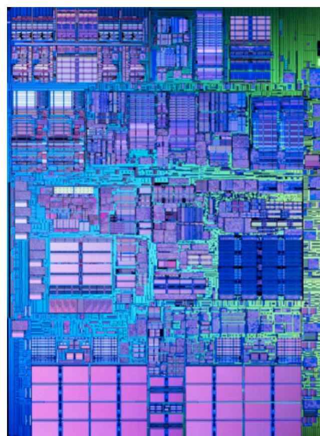
© Austin & Papaefthymiou – 2009

Example

- Consider out-of-order superscalar pipeline with 1 adder and 1 multiplier
- How can the following instructions be scheduled to minimize execution time? (Assume that an add takes 1 cycle, whereas a multiply takes 3 cycles.)

```
add  1 2 3
mult 2 3 4
add  5 6 7
add  3 4 4
add  7 8 9
add  7 7 7
```


PowerPC G5 chip



The PowerPC G5 processor

- 64-bit architecture
- Out of order execution
- 7 pipelined functional units
  - 2 load/store units
  - 2 64-bit integer units
  - 2 64-bit floating-point units
  - 1 vector unit
- 32 general 64-bit integer registers
- 32 64-bit floating-point registers
- Fixed 32-bit instruction size

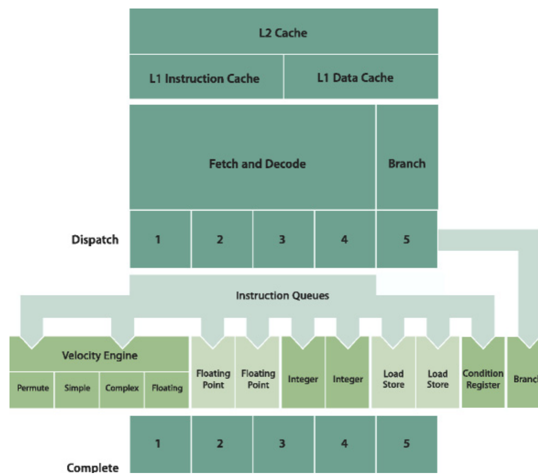
© Austin & Papaefthymiou – 2009

PowerPC G5 (continued)

- Aggressive superscalar design
- Fetches 8 32-bit instructions at once
- IPC up to 5 (for ideal code)
- IPC = 1/CPI; thus CPI as low as 0.2
- Can simultaneously sustain:
  - 4 data instructions
  - 1 branch instruction

© Austin & Papaefthymiou – 2009

G5 Architecture



Processor Comparison

Architecture	LC-2	MIPS	IA-32	IA-64	PPC-G5	AMD-64
Memory address size (bits)	16	32	32	64	64	64
Number of integer registers	8	32	8	128	32	16
Integer register size (bits)	32	32	32	64	64	64
Number of floating point registers	0	32	8	128	32	8
Floating point register size (bits)	-	32	80	82	64	80
Instruction size (bits)	32	32	8 to 136	41	32	8 to 120
Maximum clock speed, GHz (as of Feb. 2006)	?	?	3.8	1.66	2.5	2.8
Maximum instructions in flight	?	?	126	48	215	72

New Topic - Memory

- We have discussed two structures that hold data:
  - Register file (little array of storage)
  - Memory (bigger array of storage)
- We have discussed several methods of implementing storage devices:
  - Static memory (made with logic gates)
  - Dynamic memory (transistor and capacitor)
  - ROM, and other ROM-like storage (diodes)

Memory Hierarchy

- We want to have lots of memory for our processor:
  - LC2 needs 2^{16} words of memory
  - MIPS needs 2^{32} bytes of memory
  - Athlon-64 or EM64T needs 2^{64} bytes of memory
- What are our choices?
  - SRAM, DRAM, Disk, paper?

Option 1: Build It Out of Fast SRAM

- About 5 ns access
  - Decoders are big
  - Array is big
- It will cost LOTS of money
  - SRAM costs about $10 per megabyte
    - $1.25 for LC2
    - $40,960 for MIPS
    - $175 trillion for Athlon-64

© Austin & Papaefthymiou – 2009

Option 2: Build It Out of DRAM

- About 40 ns access
  - Why build a fast processor that stalls for dozens of cycles on each memory load?
- Still costs lots of money for new machines
  - DRAM costs $0.03 per megabyte
    - < $0.01 for LC2
    - $120 for MIPS/Pentium-IV/Athlon-XP
    - $500 billion for Alpha/G5/Itanium/Athlon-64

© Austin & Papaefthymiou – 2009

Option 3: Build It Using Disks

- About 3,000,000 ns access (snore!)
  - We could have stopped with the Intel 4004
- Costs are pretty reasonable
  - Disk storage costs $0.0002 per megabyte
    - Basically free for LC2
    - $1 for MIPS
    - $4 billion for Athlon-64

© Austin & Papaefthymiou – 2009

Option 4: Build It Using Tapes

- About 100,000,000,000 ns access
  - Time to load tape and wind it to the correct position
  - Faster than chiseling it on a stone tablet
- Costs are pretty reasonable
  - Tape storage costs $0.0001 per megabyte
    - Basically free for LC2
    - 50 cents for MIPS
    - $2 billion for Athlon-64

© Austin & Papaefthymiou – 2009

Option 5: Build It Using Optical Disks

- About 100,000,000,000 ns access
  - Depends mostly on speed of finding and loading media
- DVD-R/DVD+R is cheapest storage per bit available today
  - DVD-R/DVD+R media costs $0.00004 per megabyte
    - Basically free for LC2
    - $0.16 for MIPS
    - $700 million for Athlon-64 (ouch!)

Our Requirements

- We want a memory system that runs at processor clock speed (about 1 ns access)
- We want a memory system that we can afford (maybe 25% to 33% of the total system costs).
- Options 1-5 are too slow
- Options 1-2 (or 1-5) are too expensive

Time for option 6!

Option 6: Use a Little of Everything (Wisely)

- Use a small array of SRAM
  - Small means fast and cheap!
- Use a larger amount of DRAM
  - And hope that you rarely have to use it
- Use a really big amount of disk storage
  - Disks are getting cheaper at a faster rate than we fill them up
  - Everything you will read in your lifetime would easily fit on one disk
- Use tapes or optical disks to back up disks
  - Disks can fail, and you don't want to lose everything
  - Everything you will write in your lifetime would easily fit on one CD
- Don't try to buy 2^{64} bytes of anything
  - It would take years to format it anyway!
  - About 2^{45} bytes would hold all the texts in the US library of congress

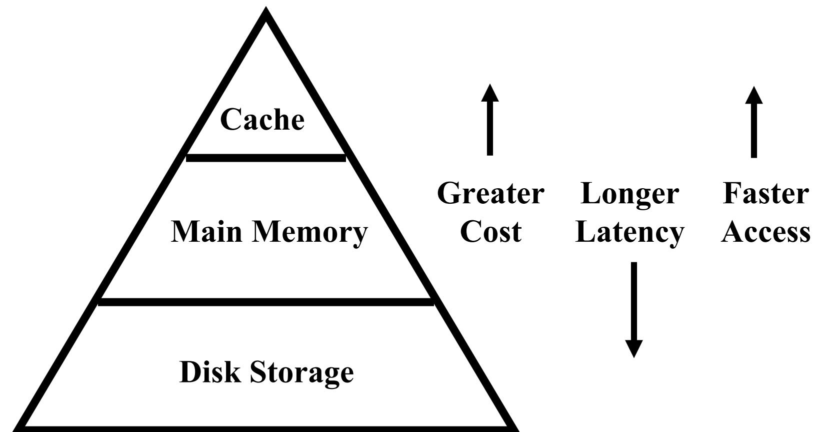
© Austin & Papaefthymiou – 2009

Option 6: The Memory Hierarchy

- Use a small array of SRAM
  - For the **CACHE** (hopefully for most accesses)
- Use a bigger amount of DRAM
  - For the **Main memory**
- Use a really big amount of Disk storage
  - For the **Virtual memory**
- Don't try to buy 2^{64} bytes of anything
  - Common sense!

© Austin & Papaefthymiou – 2009

Famous Picture of ~~Food~~ Memory Hierarchy



© Austin & Papaefthymiou – 2009

Rehashing our Terms

- The **Architectural** view of memory is:
  - What the machine language (or programmer) sees
  - Memory is just a big array of storage
- Breaking up the memory system into different pieces – cache, main memory (made up of DRAM) and Disk storage – is **not architectural**.
  - The machine language doesn't know about it
  - A new implementation may not break it up into the same pieces (or break it up at all).

© Austin & Papaefthymiou – 2009

Function of the Cache

- The cache will hold the data that we think is most likely to be referenced.
  - Because we want to maximize the number of references that are serviced by the cache to minimize the **average memory access latency**
  - How do we decide what the most likely accessed memory locations are??

© Austin & Papaefthymiou – 2009

Cache Analogy

- Hungry! Must eat!
  - **Option 1: go to refrigerator**
    - Found → eat!
    - Latency = 1 minute
  - **Option 2: go to store**
    - Found → purchase, take home, eat!
    - Latency = 20-30 minutes
  - **Option 3: grow food!**
    - Plant, wait … wait … wait … , harvest, eat!
    - Latency = ~250,000 minutes (~ 6 months)

© Austin & Papaefthymiou – 2009

Class Problem

Given the following:

Cache: 1 cycle access time
Main memory: 100 cycle access time
Disk: 10000 cycles access time

What is the average access time for 100 memory references if you measure that 90% of the cache accesses are hits and 80% of the accesses to main memory are hits.

Basic Cache Design

- Cache memory can copy data from any part of main memory
  - It has 2 parts:
    - The TAG (CAM) holds the memory address
    - The BLOCK (SRAM) holds the memory data
- Accessing the cache:
  - Compare the reference address with the tag
    - If they match, get the data from the cache block
    - If they don't match, get the data from main memory

CAMs: Content Addressable Memories

- Instead of thinking of memory as an array of data indexed by a memory address…
- Think of memory as a set of data matching a query.
  - Instead of an address, we send data to the memory, asking if any location contains that data.
  - Memory answers: yes/no (hit/miss for caches)

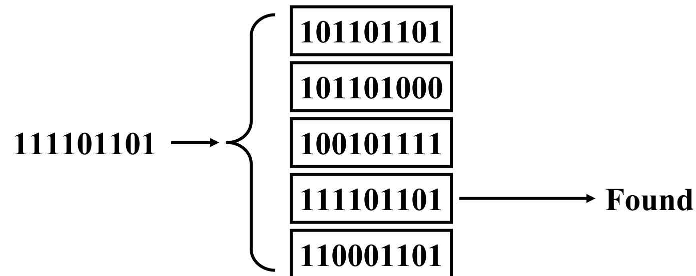
© Austin & Papaefthymiou – 2009

Operations on CAMs

- Search: the primary way to access a CAM
  - Send data to CAM memory
  - Return "found" or "not found"
  - Alternatively, return the address of where it was found
- Write:
  - Send data for CAM to remember
    - Where should it be stored if CAM is full?
      - Replacement policy
        - Replace oldest data in the CAM
        - Replace least recently searched data

© Austin & Papaefthymiou – 2009

CAM Array



5 storage element CAM array of 9 bits each

Previous Use of CAMs

- You have seen a simple CAM used before. When?

M U X

1

target

Inst mem

=?

bpc target

REG file

sign ext

Control

M U X

A L U

Data memory

M U X

beq

IF/ ID

ID/ EX

EX/ Mem

Mem/ WB

Cache Organization

- A cache memory consists of multiple tag/block pairs (called cache lines)
  - Searches can be done in parallel (within reason)
  - At most one tag will match
- If there is a tag match, it is a cache HIT
- If there is no tag match, it is a cache MISS

Our goal is to keep the data we think will be accessed in the near future in the cache

© Austin & Papaefthymiou – 2009

Cache Operation

- Every cache miss will get the data from memory and ALLOCATE a cache line to put the data in.
  - Just like any CAM write
- Which line should be allocated?
  - Random? OK, but hard to grade test questions
  - Better than random? How?

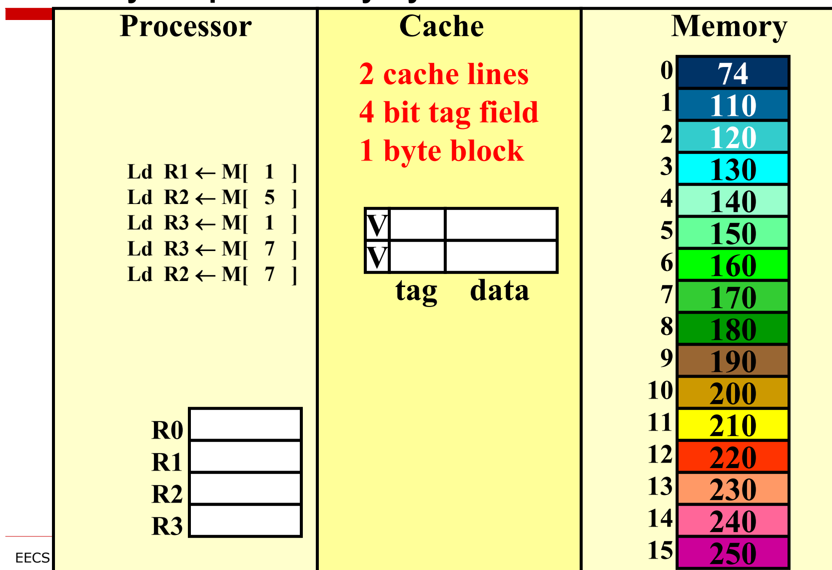
Picking the Most Likely Addresses

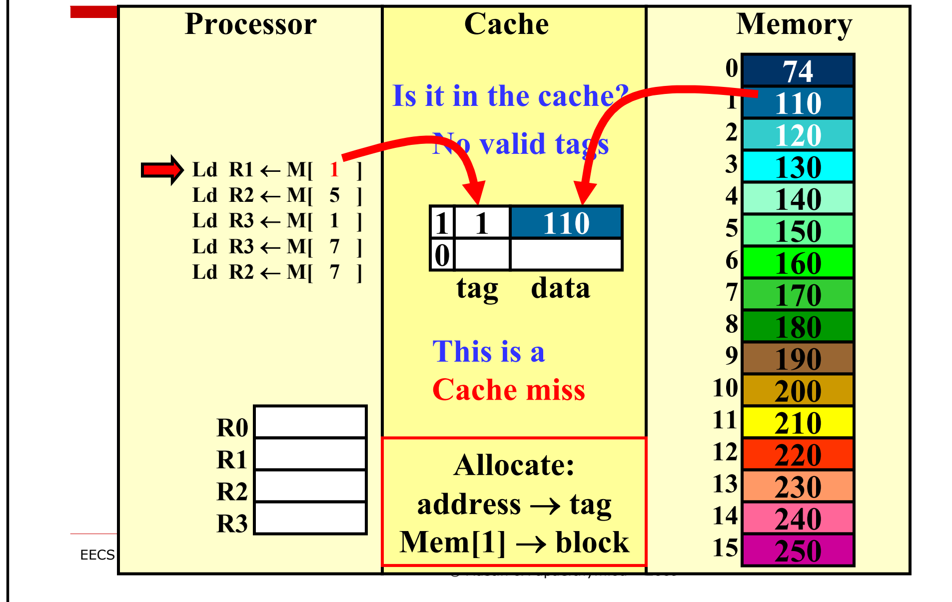
- What is the probability of accessing a random memory location?
  - With no information, it is just as likely as any other address
- But programs are not random
  - They tend to use the same memory locations over and over.
  - We can use this to pick the most referenced locations to put into the cache

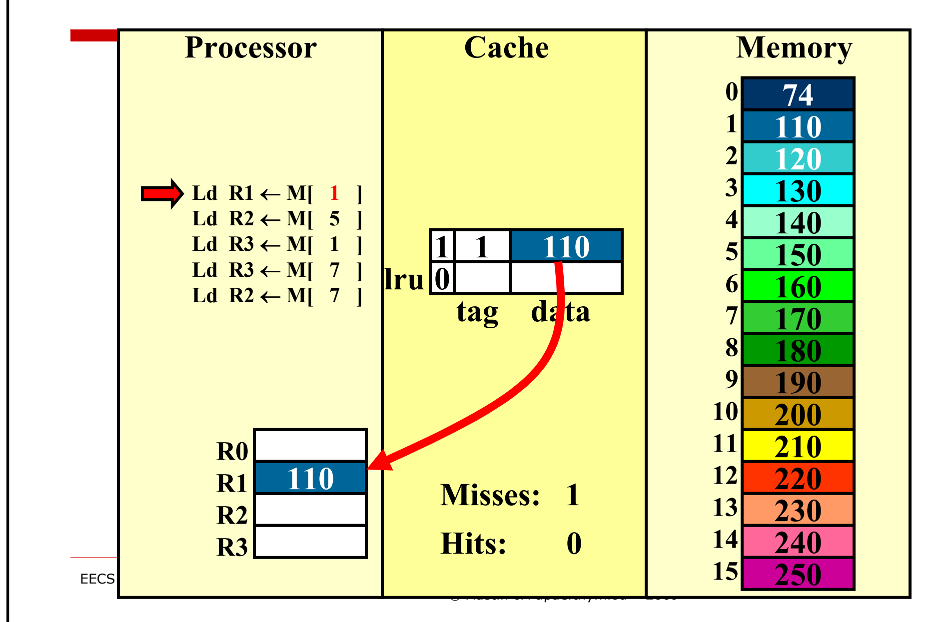
Temporal Locality

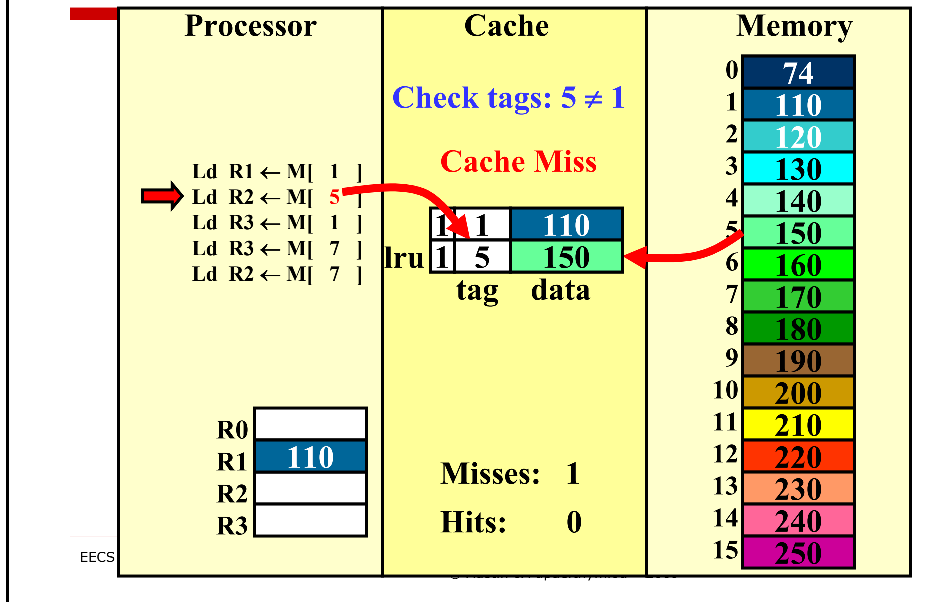
- The principle of temporal locality in program references says that if you access a memory location (e.g., 1000) you will be more likely to re-access that location than you will be to reference some other random location.
- Temporal locality says any miss data should be placed into the cache
  - It is the most recent reference location
- Temporal locality says that the least recently referenced (or least recently used – LRU) cache line should be evicted to make room for the new line.
  - Because the re-access probability falls over time as a cache line isn't referenced, the LRU line is least likely to be re-referenced.

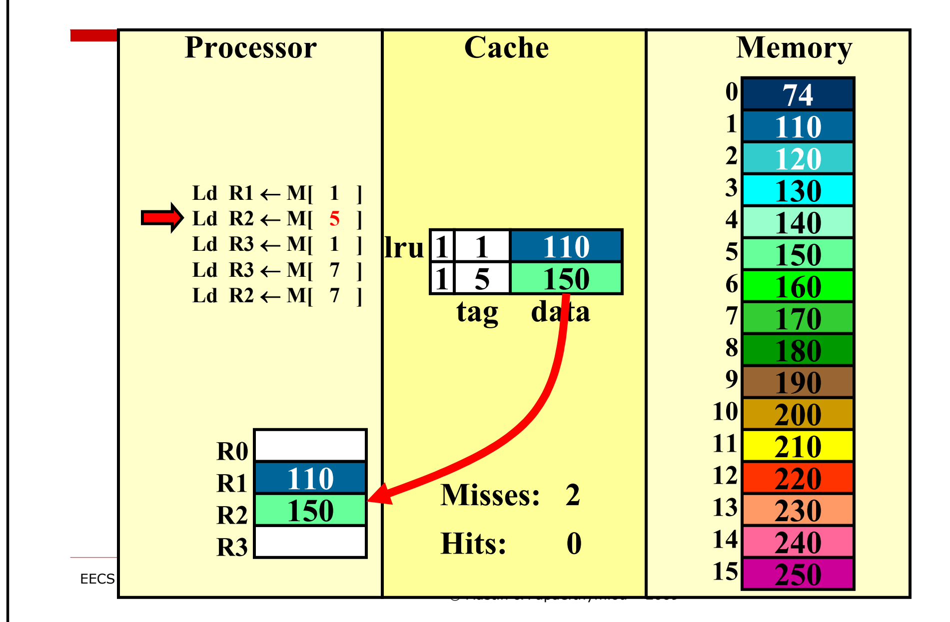
A Very Simple Memory System

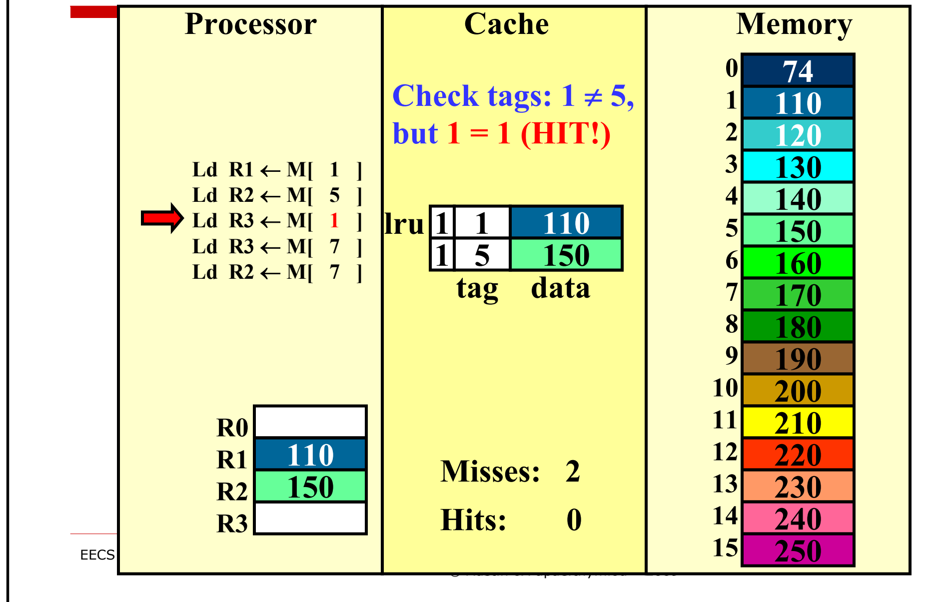


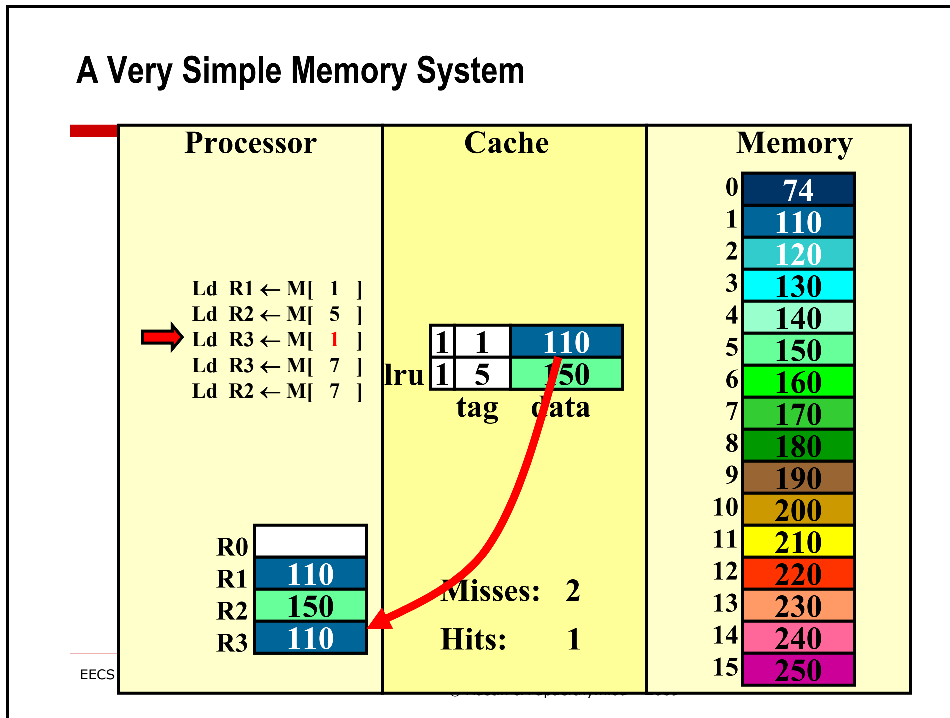

A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
Is it in the cache?
No valid tags
1 1 110
0
tag data
This is a
Cache miss
Allocate:
address → tag
Mem[1] → block
R0
R1
R2
R3
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS

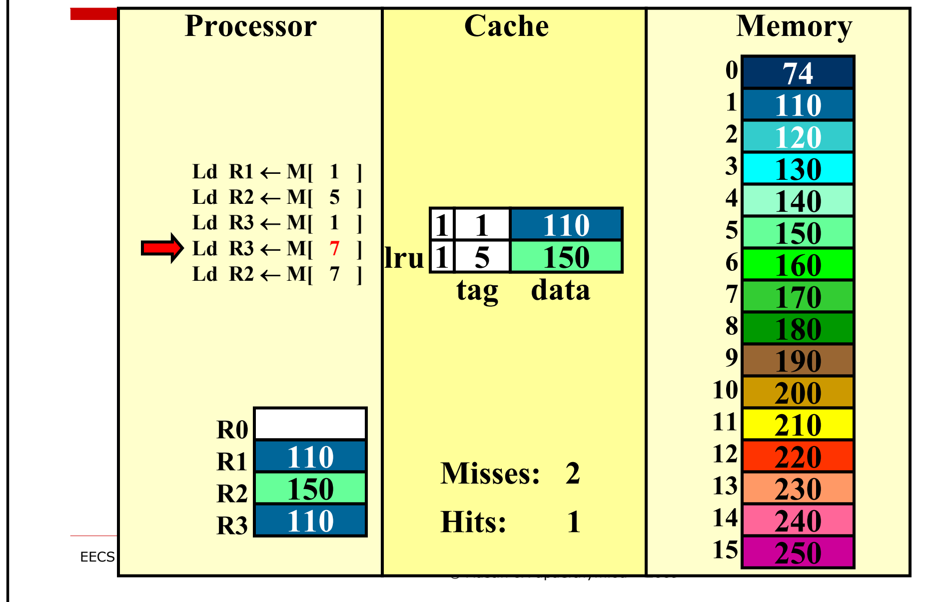

A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
1 1 110
lru 0
tag data
R0
R1 110
R2
R3
Misses: 1
Hits: 0
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS

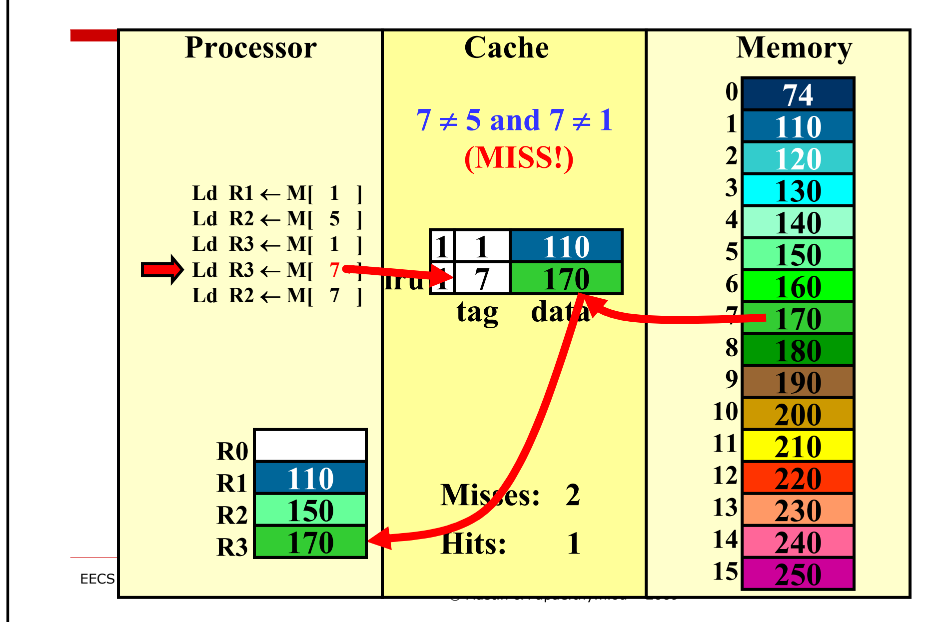

A Very Simple Memory System
Processor
Cache
Memory
Check tags: 5 ≠ 1
Cache Miss
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
tag
data
R0
R1 110
R2
R3
Misses: 1
Hits: 0
EECS

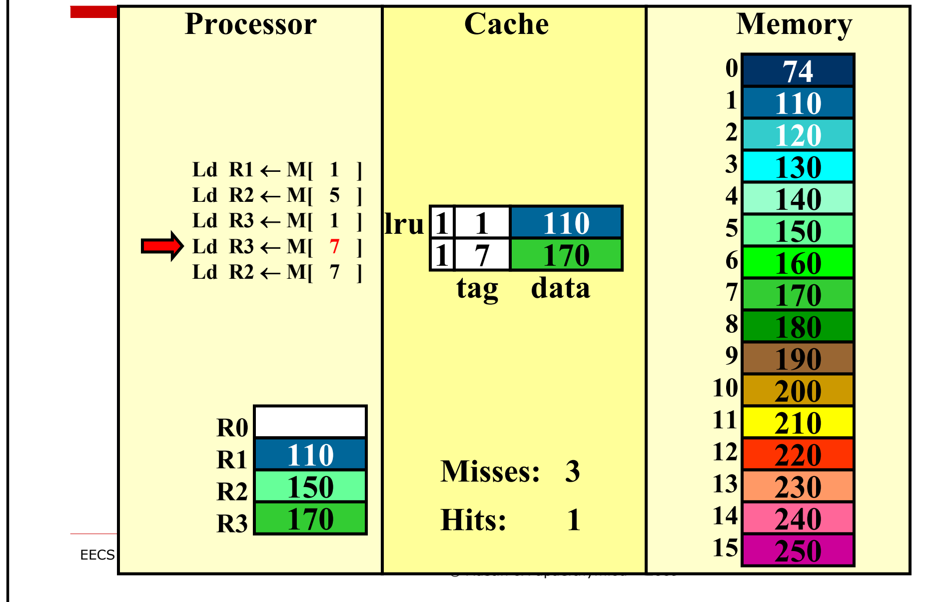

A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
tag
data
R0
R1 110
R2 150
R3
Misses: 2
Hits: 0
EECS

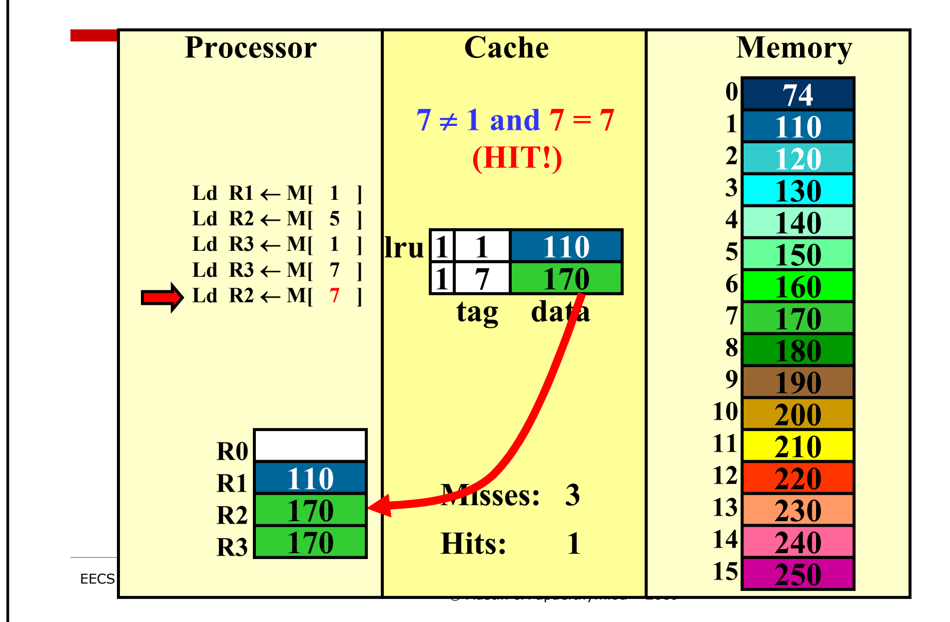

A Very Simple Memory System
Processor
Cache
Memory
Check tags: 1 ≠ 5,
but 1 = 1 (HIT!)
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
1 1 110
1 5 150
tag data
R0
R1 110
R2 150
R3
Misses: 2
Hits: 0
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS


A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
1 1 110
1 5 150
tag data
R0
R1 110
R2 150
R3 110
Misses: 2
Hits: 1
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS

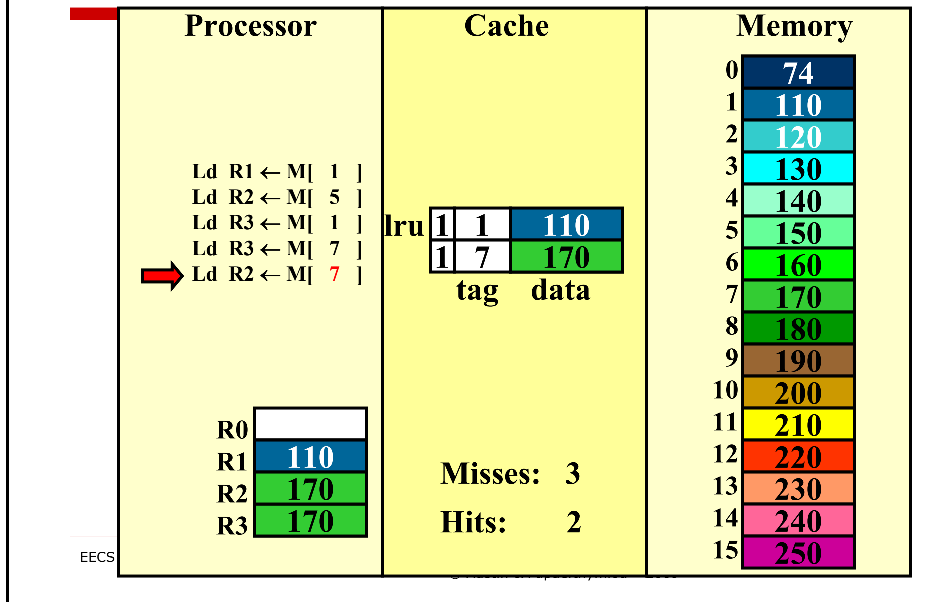

A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
1 1 110
lru 1 5 150
tag data
R0
R1 110
R2 150
R3 110
Misses: 2
Hits: 1
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS


A Very Simple Memory System
Processor
Cache
Memory
7 ≠ 5 and 7 ≠ 1
(MISS!)
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
1 1 110
lru 1 7 170
tag data
R0
R1 110
R2 150
R3 170
Misses: 2
Hits: 1
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS


A Very Simple Memory System
Processor
Cache
Memory
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
1 1 110
1 7 170
tag data
R0
R1 110
R2 150
R3 170
Misses: 3
Hits: 1
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS


A Very Simple Memory System
Processor
Cache
Memory
7 ≠ 1 and 7 = 7
(HIT!)
Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]
lru
1 1 110
1 7 170
tag data
R0
R1 110
R2 170
R3 170
Misses: 3
Hits: 1
0 74
1 110
2 120
3 130
4 140
5 150
6 160
7 170
8 180
9 190
10 200
11 210
12 220
13 230
14 240
15 250
EECS

A Very Simple Memory System



Something To Think About

- Does an optimal replacement policy exist?
  - That is, given a choice of cache lines to replace, which one will result in the fewest total misses during program execution
  - Hint: a crystal ball will come in handy in solving this problem…
- Why would we care?

© Austin & Papaefthymiou – 2009