CDA5106 - Advanced Computer Architecture
Final Exam Review

Kobee Raveendran

1 Module 1: High-Performance Microprocessor Architecture

1.1 Module 1.2: Power Wall and Dennard Scaling

1.1.1 Notes

- energy: ability of a physical system to do work on other physical systems (unit: joule)
- power: rate at which energy is transformed (unit: watt; 1 watt = 1 joule delivered per second)
  - power = $V \cdot I$ (V = voltage, I = current)
- for capacitors:
  - energy stored = $0.5 \cdot C \cdot V^2$ (C = capacitance, V = voltage)
  - if a capacitor is drained at a frequency of f per second: power = $\frac{energy}{second} = 2 \cdot 0.5CV^2 = CV^2$
- Power wall problem
  - $P_{dyn} = ACV^2f$
  - A: fraction of gates actively switching
  - C: total capacitance of all gates
  - V: supply voltage
  - f: frequency of switching
- Power wall fundamentals
  - max frequency vs. threshold voltage:
  - $f_{max} = c \cdot \frac{(V - V_{thd})^{1.3}}{V}$
- **Dennard Scaling Example (old)**
  - if gate length (transistor size) scales by $S = 0.7$ (both length and width), then:
  - capacitance scales by $S = 0.7$
  - original area scales by $S^2 = 0.5$
  - number of transistors scales by $\frac{1}{S^2} \approx 2$
  - supply voltage (V) scales by $S = 0.7$
  - frequency (f) scales by $\frac{1}{S} = 1.4$
  - then, **dynamic power** $P_{dyn} = ACV^2f$
  - and **new dynamic power** $P'_{dyn} = A'C'V'^2f'$
  - $P'_{dyn} = (2A)(0.7C)(0.7V)^2(1.4f) \approx 1 \cdot ACV^2f = P_{dyn}$
- **Post Dennard Scaling example (new)**
  - capacitance scales by $S = 0.7$
  - number of transistors scales by $\frac{1}{S^2} = 2$

- supply voltage (V) cannot scale without also scaling threshold voltage (V_{thd}), and doing that increases static power exponentially
- frequency (f) scales by $\frac{1}{S} = 1.4$
- result: dynamic power doubles every generation
- $P_{dyn} = ACV^2 f$
- $P'_{dyn} = A'C'V'^2 f' = (2A)(0.7C)(1 \cdot V)^2(1.4f) \approx 2 \cdot P_{dyn}$

1.1.2 Exercises

1. Suppose that instead of progressing at a ratio of 0.7, Moore's law slows down and transistor gate length scales at a ratio of 0.8 instead. Find the dynamic power consumption under *unlimited* and *limited* scaling for the next process generation.
    - Unlimited/old scaling rule
        * gate length scales by $S = 0.8$
        * capacitance scales by $S = 0.8$
        * original area scales by $S^2 = 0.64$
        * num transistors thus scales by $\frac{1}{S^2} = 1.56$
        * supply voltage scales by $S = 0.8$
        * frequency scales by $\frac{1}{S} = 1.25$
        * dynamic power stays constant:
          $P_{dyn} = (1.56A)(0.8C)(0.8V)^2(1.25f)$
    - leakage-limited/new scaling
        * capacitance scales by $S = 0.8$
        * num transistors scales by $\frac{1}{S^2} = 1.56$
        * supply voltage does not scale without scaling threshold voltage too, which increases static power exponentially
        * frequency scales by $\frac{1}{S}1.25$
        * dynamic power consumption increases:
          $P'_{dyn} = (1.56A)(0.8C)(V)^2(1.25f) = 1.56 \cdot P_{dyn}$
2. With limited voltage scaling, suppose that we want to keep the dynamic power consumption constant in the next generation by keeping frequency constant and reduce die area. How much should we reduce die area to achieve that?
    - gate length scales by $S = 0.7$
    - capacitance scales by $S = 0.7$
    - original area scales by $S^2 = 0.5$
    - supply voltage and frequency are constant
    - dynamic power consumption must stay constant: $P'_{dyn} = P_{dyn}$
      $ACV^2 f = A'(0.7C)V^2 f \longrightarrow A = 0.7A'$
    - number of transistors in the next generation: $A' = 1.4A$ (instead of 2A like before; i.e. 70% of 2A)
    - thus die area shrinks by 30%
3. Describe the difference between energy and power.

   Power is the rate of energy consumption.
4. Describe the impact of threshold voltage choice on static and dynamic power consumption as transistors are scaled down.

   If threshold voltage is lowered, dynamic power decreases (nearly linearly) but static power increases exponentially.
5. How has processor design adapted to the power wall problem?

   Stalling frequency growth, multicore, and sophisticated power management (clock gating, voltage and frequency scaling, power gating).

1.1.3 Overview of ILP Techniques

Caches example

- processor with 1-ns clock
- 64KB cache memory with 2-ns read time, 95% hitrate
- 512MB main memory with 150-ns read time
- What is the average access time (AAT) in this memory system?

Answer:

- hits: $95 \cdot 2$ ns, misses: $5 \cdot (2 + 150)$ ns
- total = hit time + miss time = $190 + (10 + 750) = 950ns$
- AAT = $\frac{total}{100} = 9.5ns$

2 Module 2: Performance, Cost, and Reliability of Microprocessors

2.1 Performance Evaluation 1

2.1.1 Amdahl's Law

- performance improvement ("speedup") is limited by the part you can't improve
- (s) $Speedup_{enhanced}$ = best case speedup from gizmo alone
- (f) $Fraction_{enhanced}$ = fraction of task that gizmo can enhance
- $s_{overall} = \frac{1}{(1-f)+\frac{f}{s}}$

Example:

- jet plane wing simulation, where 1 run takes 1 week on your computer
- your program is 80% parallelizable
- new supercomputer has 100,000 processors
- $s = 100,000$
- $f = 0.8$
- overall speedup: $s_{overall} = \frac{1}{(1-f)+\frac{f}{s}} = \frac{1}{(1-0.8)+\frac{0.8}{100000}} \approx \frac{1}{0.2} = 5$
- only about 5 times faster (33 hours instead of 1 week), but not worth the high price tag (using a cheaper computer with only 100 processors instead yields a 4.8X speedup!)

More examples:
Ex 1:

- $f = 0.95$
- $s = 1.10$
- $s_{overall} = \frac{1}{(1-0.95)+\frac{0.95}{1.10}} = 1.094 \approx 1.10$

Ex 2:

- $f = 0.05$
- $s \to \infty$
- $s_{overall} = 1.053$

2.1.2 Run Time

- CPU time = clock cycle count × cycle time
- cycles per instruction (CPI) = $\frac{\texttt{clock cycle count}}{\texttt{instruction count}}$
- CPU time = IC × CPI × CT

2.2 Performance Evaluation 2

Determine speedup by comparing program times with respect to a reference machine.

- arithmetic mean (which one should we trust?):

	Computer A	Computer B	B vs. A
Program P1	2X faster	4X faster	2X faster
Program P2	5X faster	15X faster	3X faster
Average	3.5X	9.5X	

  Speedups:

  - method 1: program-wise ⟶ $\frac{2+3}{2}$ = 2.5X faster
  - method 2: machine-wise ⟶ $\frac{9.5}{3.5}$ = 2.71X faster

- geometric mean (consistent):

$$gmean = \sqrt[n]{\prod_{i=1}^{n}} = exp(\frac{\frac{1}{n}\sum_{i=1}^{n}\ln(x_i)}{n})$$

	Computer A	Computer B	B vs. A
Program P1	2X faster	4X faster	2X faster
Program P2	5X faster	15X faster	3X faster
Average	$\sqrt{10}$	$\sqrt{60}$	$\sqrt{6}$

  Speedups:

  - method 1: program-wise ⟶ B is $\sqrt{2 \cdot 3} = \sqrt{6}$X faster
  - method 2: machine-wise ⟶ B is $\sqrt{60} \cdot \sqrt{10} = \sqrt{6}$X faster

- (also important): geometric standard deviation

$$gstdev = exp(\sqrt{\frac{\prod_{i=1}^{n}(\ln x_i - \ln gmean)^2}{n}})$$

  in plain English: for each "component" speedup vs. ref machine, take its natural log and subtract the natural log of the gmean from that. Square it and multiply all of these together, then divide by n. Finally take the square root of this, then take e to the power of the result.

2.2.1 Exercises

Given the following table of speedups for machines A and B relative to a reference machine:

Prog	X (secs)	A (secs)	B (secs)
App 1	30	15	10
App 2	20	15	10
App 3	40	20	30
App 4	15	20	15

Compute the following (see post-computation table below to find them all):

- geometric speedup of machine A vs. base machine X

  from the table, we find that A has a 1.41X speedup over X

- geometric speedup of machine B vs. base machine X

  from the table, we find that B has a 1.68X speedup over X

- geometric speedup of machine B vs. machine A

  from the table, we find that B has a 1.19X speedup over A

- geometric standard deviation of the speedup of machine A over machine X

  $gstd = exp(\sqrt{\frac{1}{4} \cdot \ln^2(\frac{2}{1.41}) \ln^2(\frac{1.33}{1.41}) \ln^2(\frac{2}{1.41}) \ln^2(\frac{0.75}{1.41})})$

  $gstd = 1.002255... \approx 1$

Prog	A vs. X	B vs. X	B vs. A
App 1	2X	3X	1.5X
App 2	1.33X	2X	1.5X
App 3	2X	1.33X	0.67X
App 4	0.75X	1X	1.33X
Product	4X	8X	2X
gmean	1.41X	1.68X	**1.19X**

2.3 Cost and Reliability

2.3.1 Failure Rates (λ)

- λ = the number of failures that occur per unit time in a component/system
- FIT (failure in time) = number of failures in 10^9 hours
- example: 10,000 microprocessor chips used for 1,000 hours, and 8 of them fail. Failure rate is thus $\frac{8}{10{,}000 \cdot 1{,}000} = 8 \cdot 10^{-7}$ (failures per hour per chip) $\cdot 10^9$ hours = 800 FITs

2.3.2 Reliability Metrics

- $R(t)$ = probability that the system still works correctly at time t
- $W_N(t)$ = number of items (of the same kind) that would still be working at time t
- if λ is constant, then $R(t) = e^{-\lambda t}$
- Mean Time Between Failure (MTBF) = $\frac{1}{\lambda}$

2.3.3 System Reliability

Assume that:

- M components are in the system with failure rates $\lambda_1, \lambda_2, ..., \lambda_m$
- for the system to work properly, all components must also work properly
- a component's reliability is independent of any other component's reliability
- then, system failure rate = sum of component's failure rates
- $R_{sys}(t) = R_1(t) \cdot R_2(t) \cdot ... \cdot R_m(t) = e^{-\lambda_1 t \cdot ... \cdot -\lambda_m t} = e^{-(\lambda_1 + \lambda_2 + ... + \lambda_m)t} = e^{-\lambda_{sys} t}$

Other metrics:

- Mean Time To Repair (MTTR): mean time to repair/recover from a fault
- Mean Time Between Failure (MTBF): mean time between 2 consecutive failures
- if each failure is repaired, then MTBF = MTTF + MTTR
- usually, MTTF $\gg$ MTTR, so MTBF and MTTF are often interchangeable

2.3.4 Examples

Assume a disk subsystem with:

- 10 disks each rated at 10^6-hour MTTF
- 1 SCSI controller rated at $5 \cdot 10^5$-hour MTTF
- 1 power supply rated at $2 \cdot 10^5$-hour MTTF
- 1 fan rated at $2 \cdot 10^5$-hour MTTF
- 1 SCSI cable rated at 10^6-hour MTTF

Find the failure rate of the entire disk subsystem.
$R_{sys}(t) = 10 \cdot \frac{1}{10^6} + \frac{1}{5 \cdot 10^5} + \frac{2}{2 \cdot 10^5} + \frac{1}{10^6} = \frac{10+2+5+5+1}{10^6} = \frac{23}{10^6} = \frac{23{,}000}{10^9} = 23{,}000$ FIT
Thus, MTTF $= \frac{1}{\lambda_{sys}} = \frac{1}{23{,}000} \cdot 10^9 \approx 43{,}500$ hours

3 Instruction Set Design

3.1 Instruction Set Architecture 1

3.1.1 Styles of ISAs

- Stack:
  - `push <addr>, pop <addr>` (or ALU instructions)
  - ALU: pop two entries, perform ALU operation, push result onto stack
  - compact instruction format
    - all calculation operations take 0 operands
    - flexible; used for compiling Java bytecode (not dependent on registers in architecture [but all have stacks])
- Accumulator:
  - `load/store/ALU <addr>` (result affects accumulator register)
  - also very compact (all operations take 1 operand, the other is implicitly the accumulator register)
  - less dependence on memory than stack-based
- Register-memory:
  - `load/store/ALU <reg>, <reg/addr>`
    - at most one operand can be a memory address
    - leftmost register is the destination (if applicable)
- Load-Store:
  - `load/store <reg>, <addr>`
  - ex: `ALU <reg1>, <reg2>, <reg3>`: reg1 is destination, other 2 are source registers

Example for adding two numbers:

Stack	Accumulator	Register-memory	Load-Store
push A	load A	load R1 A	load R1 A
push B	add B	add R1 B	load R2 B
add	store C	store C R1	add R3 R1 R2
pop C			store C R3

Pros/cons:

- load-store
  - (+) fixed length instructions possible (allows for easy fetch/decode)
  - (+) simpler hardware: efficient pipeline and potentially lower cycle time
  - (-) higher instruction count

  - (-) fixed-length instructions can be wasteful (more bits than needed for some instructions)

- register-memory
  - (+) no need for extra loads
  - (+) better usage of bits (these pros lead to better code density)
  - (-) destroys source operand(s) (i.e. `add R1 R2`)
  - (-) may impact cycles per instruction

- memory-memory
  - (+) most compact (code density)
  - (-) high memory traffic (thus bottlenecked by memory)

3.2 Instruction Set Architecture 2

3.2.1 RISC and Common Addressing Modes

- register
  - `add R4 R3 // R4 = R4 + R3`
  - used when value is in a register

- immediate
  - `add R4 #3 // R4 = R4 + 3`
  - used for small constants (which occur frequently)

- displacement
  - `add R4 100(R1) // R4 = R4 + MEM[100 + R1]`
  - accesses the frame (arguments, local variables)
  - access the global data segment
  - accesses the fields of a data struct

- register deferred/register indirect
  - `add R3 (R1) // R3 = R3 + MEM[R1]`
  - accesses using a computed memory address

- indexed
  - `add R3 (R1 + R2) // R3 = R3 + MEM[R1 + R2]`
  - array accesses: R1 = base, R2 = index

- direct/absolute
  - `add R1 (1001) // R1 = R1 + MEM[1001]`
  - accessing global ("static") data

- memory deferred/memory indirect
  - `add R1 @(R3) // R1 = R1 + MEM[MEM[R3]]`
  - pointer dereferencing: `x = *p;` (if p is not register-allocated)

- autoincrement/postdecrement
  - `add R1 (R2)+ // R1 = R1 + MEM[R2]; R2 = R2 + d` (d is the size of the operation)
  - looping through arrays, stack pop

- autodecrement/predecrement
  - `add R1 -(R2) // R1 = R1 + MEM[R2]; R2 = R2 - d` (d is the size of the operation)

  - same uses as autoincrement, stack push

- scaled
  - `add R1 100(R2)[R3] // R1 = R1 + MEM[100 + R2 + R3 * d]`
  - array accesses for non-byte-sized elements

3.3 Instruction Set Architecture 3

3.3.1 Condition Codes (for branch instructions)

- Z
  - zero flag
  - indicates the result of an arithmetic/logical expression is zero
- C
  - carry flag
  - indicates that an operation has a carry out. Enables numbers larger than a single word to be added/subtracted
- S/N
  - sign/negative flag
  - indicates the result of an operation is negative
- V/O/W
  - overflow flag
  - indicates the result of an operation is too large to fit in a register (using 2's complement representation)

3.3.2 Instruction Encoding Tradeoffs

- variable width
  - (+) very versatile, uses memory efficiently
  - (-) instruction words must be decoded before number of bytes is known (harder to fetch/decode)
- fixed width
  - (+) every instruction word is an instruction, thus easier to fetch/decode
  - (-) uses memory inefficiently (same num. bits even for short instructions)
- hybrid
  - primarily for embedded processors to conserve memory
  - often use a subset of instructions, fewer registers, or even instruction compression

3.4 ISA Examples

3.4.1 MIPS

Characteristics:

- load/store ISA: only loads/stores can have memory operands, makes for easy pipelining and uniform instruction width
- fixed instruction width = easy fetch and decode
- small number of addressing modes = easy pipelining
- large register file: 32 integer and 32 floating point registers
- aligned memory = easy data fetching

- quantitatively designed

Instruction format:

- R: `op, rs, rt, rd, shamt, funct` (`shamt` = shift amount, `funct` = ALU function)
- I: op, rs, rt, 16-bit address
- J: op, 26-bit address

4 Memory Hierarchy

4.1 Basics of Cache Architecture

4.1.1 Cache Organization

A cache is similar to a table

- a **set** in cache = a row in table
- a **way** in cache = a column in table
- a **line** in cache = a cell in table

4.1.2 Choices on Cache Associativities

- direct mapped cache: a block can be placed in only one line in the cache (i.e. vertical array table)
  - rigid placement of blocks in cache set
  - usually has higher miss rate
  - but power efficient (no need to search an entire way if the way is only one cell)
- fully associative cache: a block can be placed in any line in the cache (i.e. a horizontal array table)
  - flexible placement of blocks in cache set
  - has the lowest miss rate
  - but power hungry (have to search entire set [aka the whole cache] to find a block)
- set-associative cache: a block can be placed in one of the ways in a set (i.e. a 2D array table)

4.1.3 Cache Parameters

- SIZE = total amount of cache data storage in bytes
- BLOCKSIZE = total number of bytes in a single block
- ASSOC = associativity (number of lines in a set)

Formulas:

# of blocks in a cache = $\frac{SIZE}{BLOCKSIZE}$

# of sets in a cache = $\frac{\texttt{\# cache blocks}}{ASSOC} = \frac{SIZE}{BLOCKSIZE \cdot ASSOC}$

# of index bits = $\log_2(\#sets)$

# of block offset bits = $\log_2(BLOCKSIZE)$

# of tag bits = $32 - \texttt{\#index bits} - \texttt{\#offset bits}$

4.1.4 Examples

Example 1: Processor accesses a 256B direct-mapped cache, which has a block size of 32B, with the below sequence of addresses. Show the contents of the cache after each access, and count the number of hits and replacements.

# index bits = $\log_2(8) = 3$
# offset bits = $\log_2(32) = 5$
# tag bits = $32 - 3 - 5 = 24$

Address (hex)	Tag (hex)	Index and offset bits (binary)	Index (decimal)	Comment
0xFF0040E0	0xFF0040	**111**0 0000	7	miss
0xBEEF005C	0XBEEF00	**010**1 1100	2	miss
0xFF0040E2	0xFF0040	**111**0 0010	7	hit
0xFF0040E8	0xFF0040	**111**0 1000	7	hit
0x00101078	0x001010	**011**1 1000	3	miss
0x002183E0	0x002183	**111**0 0000	7	miss/rep
0x00101064	0x001010	**011**0 0100	3	hit
0x0012255C	0x001225	**010**1 1100	2	miss/rep
0x00122544	0x001225	**010**0 0100	2	hit

4.1.5 Write Updates

- Write-through (WT) policy
  - writing to some level in the cache also means writing through to subsequent levels in the cache (i.e. next level in the memory hierarchy)
- Write-back (WB) policy
  - write only to the specified cache level, and set its dirty bit
  - when the block you wrote to is replaced (evicted), write the block to the next level of memory hierarchy
- Write-allocate (WA) policy
  - bring block into cache if the write misses (just like in read misses)
  - typically used in conjunction with write-back (WBWA)
- Write-no-allocate (NA) policy
  - do not bring the block into the cache if write misses
  - *must* be used in conjunction with write-through (WTNA)

4.1.6 Victim Cache

- small fully-associative cache that sits alongside the primary cache
- when main cache evicts a block, the victim cache takes the evicted block (called the "victim")
- when the main cache misses, it searches the victim cache for recently discarded blocks; a victim cache hit means the main cache doesn't have to go to memory to search for a block
- example:
  - L1 cache (initially a set contains just A)
  - 2-entry victim cache that contains X, and Y (current LRU)
  - B misses in L1, evicts A, A goes to victim cache and replaces Y (previous LRU); X is new LRU
  - A then misses in L1 but hits in victim cache, so A and B *swap* positions (A goes to L1, B goes to VC; note that X (prev LRU) is not replaced in the case of victim cache hits)
  - thus a victim cache is useful in cases of repeated conflicts and gives the illusion of set-associativity

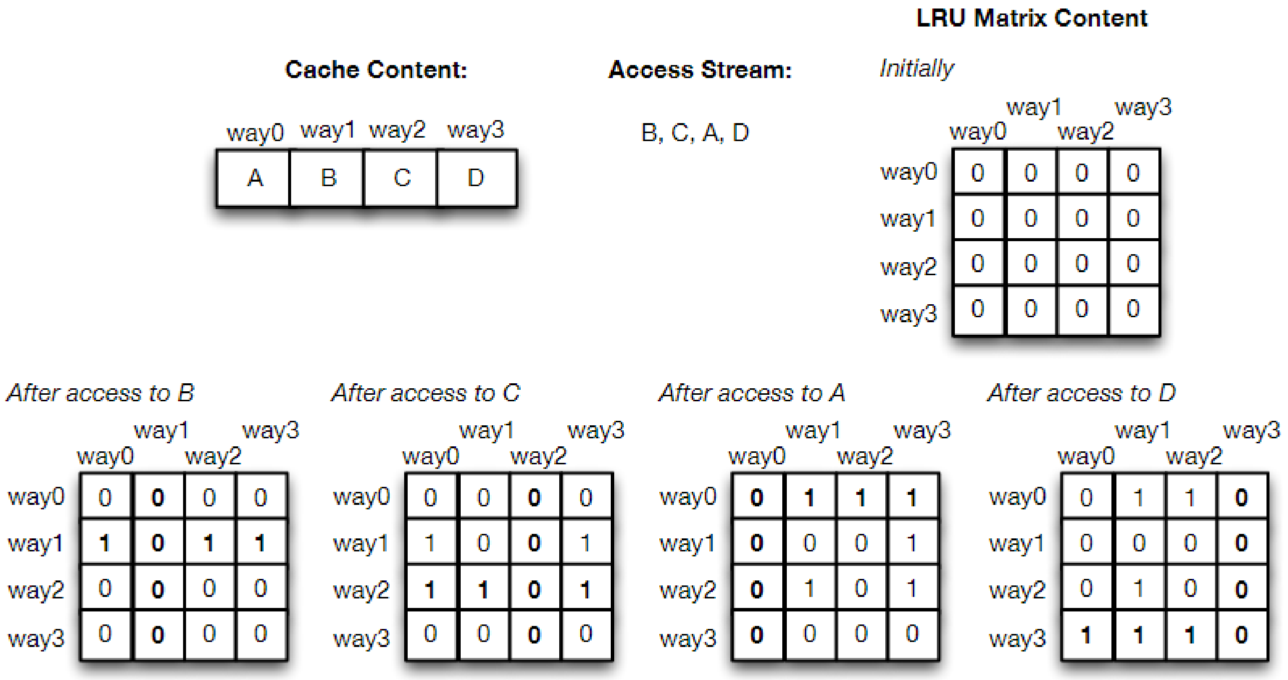
4.2 Replacement Policies

4.2.1 Optimal Replacement Policy

- look into future and determine when each block in a set is needed again (if at all)
- replace the block needed farthest in the future
- note: not practical since we don't know in advance when blocks are needed, but this is the theoretical gold standard with which other replacement policies are evaluated against

4.2.2 LRU implementation

- assign a row and column in LRU matrix to each way in the set
- if there's a hit in a way, set the row corresponding to that way, and unset the column corresponding to that way
- the number of 1's in a row specifies the MRU order (thus, the LRU is the one with all 0's)



Cache Content:

way0	way1	way2	way3
A	B	C	D

Access Stream:

B, C, A, D

LRU Matrix Content

Initially

	way0	way1	way2	way3
way0	0	0	0	0
way1	0	0	0	0
way2	0	0	0	0
way3	0	0	0	0

After access to B

	way0	way1	way2	way3
way0	0	**0**	0	0
way1	**1**	**0**	**1**	**1**
way2	0	**0**	0	0
way3	0	**0**	0	0

After access to C

	way0	way1	way2	way3
way0	0	0	**0**	0
way1	1	0	**0**	1
way2	**1**	**1**	**0**	**1**
way3	0	0	**0**	0

After access to A

	way0	way1	way2	way3
way0	**0**	**1**	**1**	**1**
way1	**0**	0	0	1
way2	**0**	1	0	1
way3	**0**	0	0	0

After access to D

	way0	way1	way2	way3
way0	0	1	1	**0**
way1	0	0	0	**0**
way2	0	1	0	**0**
way3	**1**	**1**	**1**	**0**

4.2.3 Pseudo-LRU implementation

- LRU implementation takes $O(way^2)$ space and time; too expensive
- PLRU approximates LRU with decent accuracy
- PLRU complexity is $O(way)$

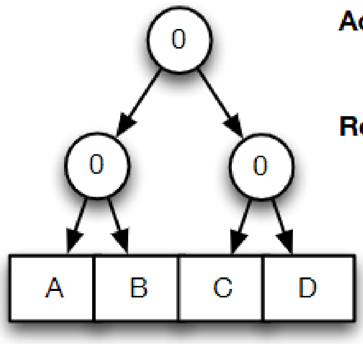
Cache Content:

way0	way1	way2	way3
A	B	C	D

Access Stream:

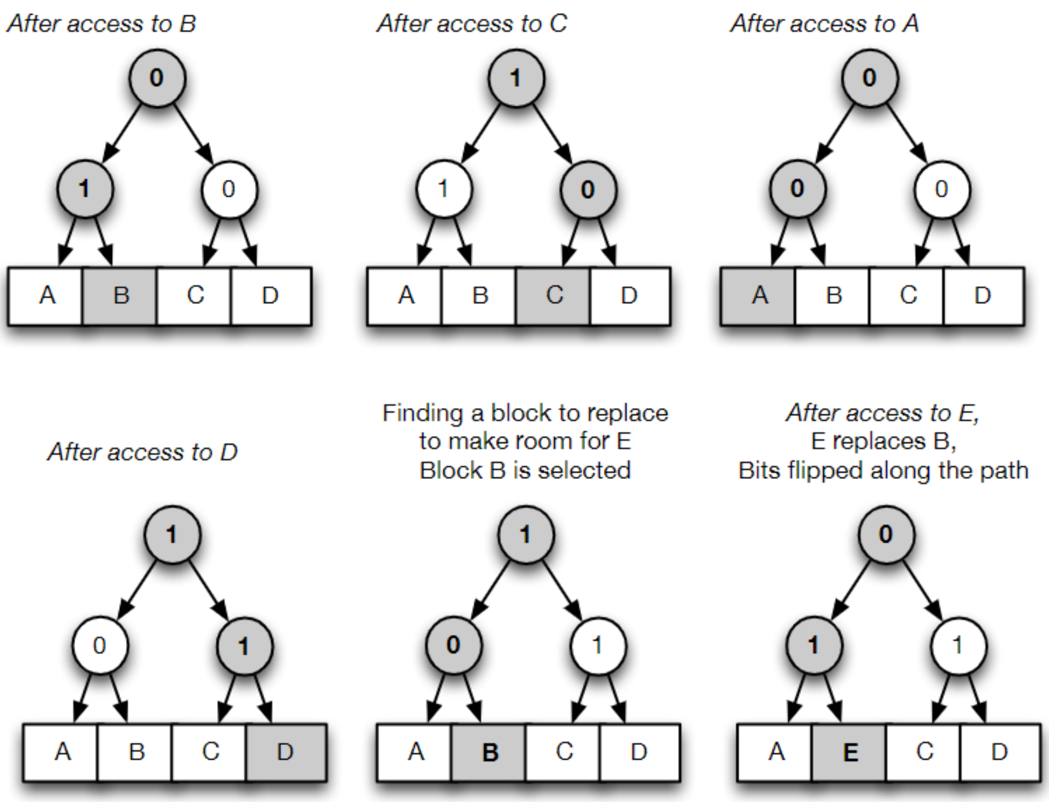
B, C, A, D, E

Pseudo LRU Tree



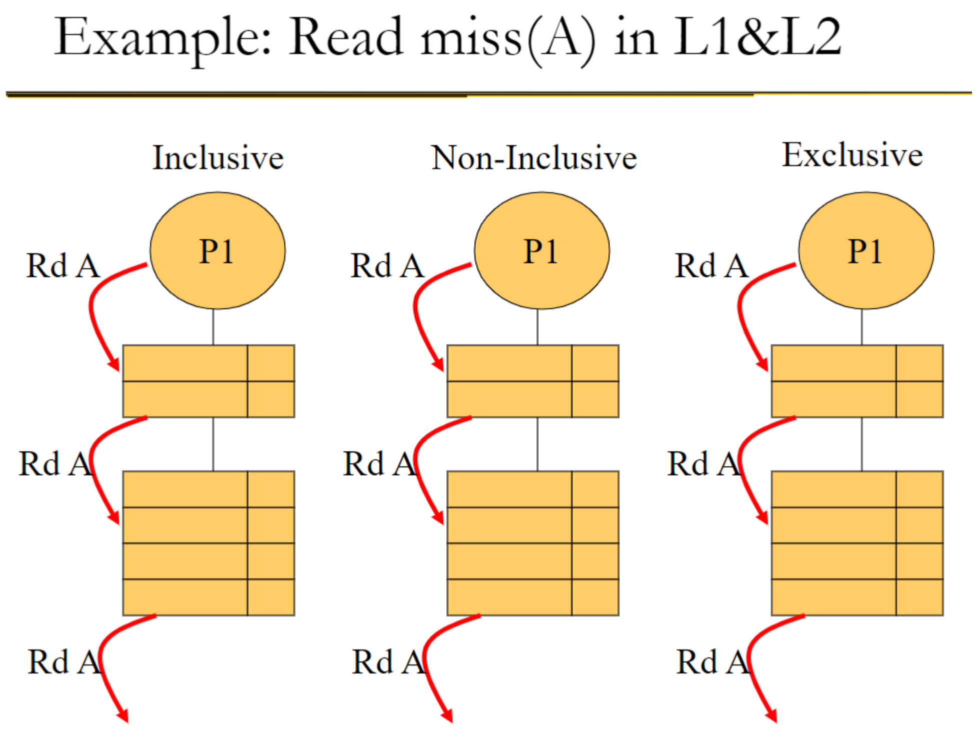
Access:
0 = Left, 1 = Right

Replacement:
0 = Right, 1 = Left

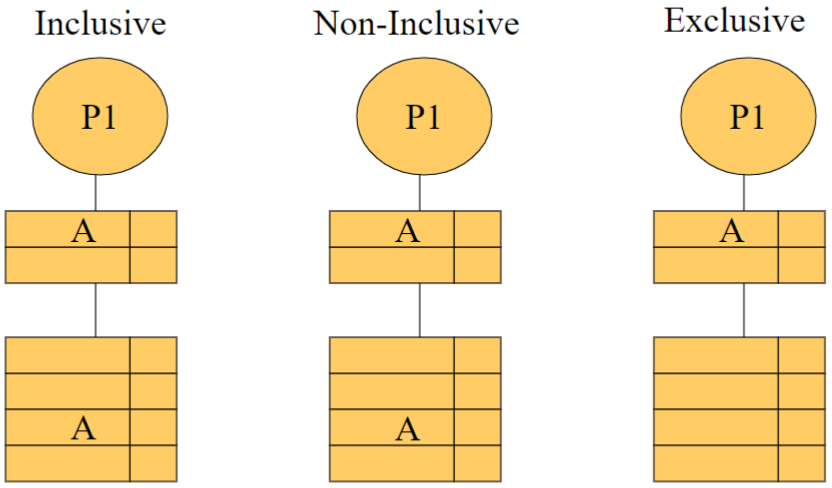


4.3 Inclusion Policies

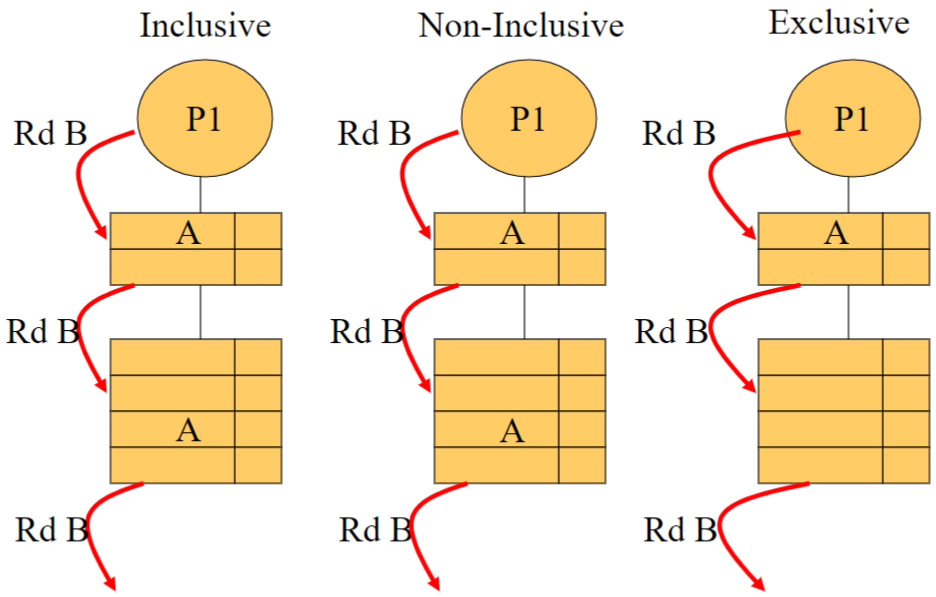
- block in inner cache level *always* included in outer level cache too: **inclusive**
- block in inner cache level *never* included in outer level cache too: **exclusive**
- block in inner cache level *sometimes* included in outer level cache: **non-inclusive**


Example: Read miss(A) in L1&L2

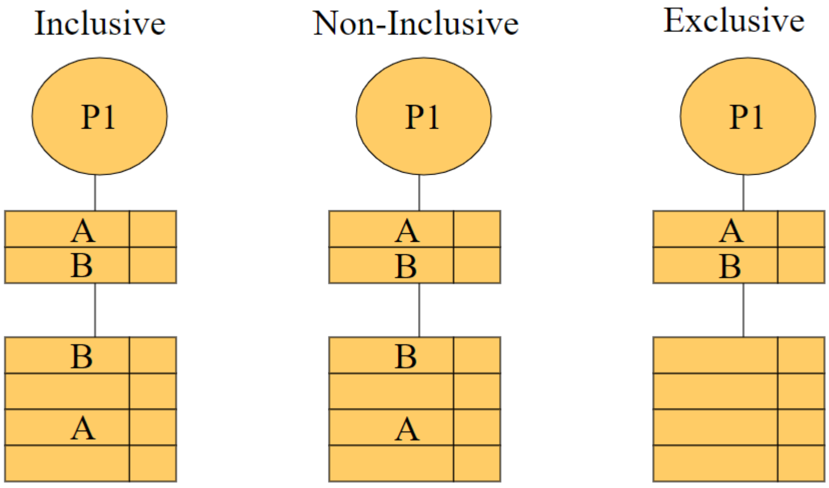
Fill at L1? Fill at L2?



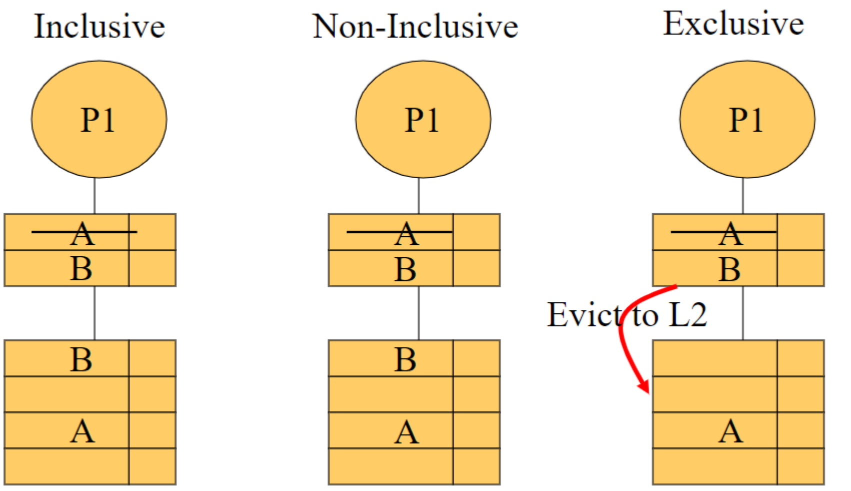
Example: Read miss(B) in L1&L2



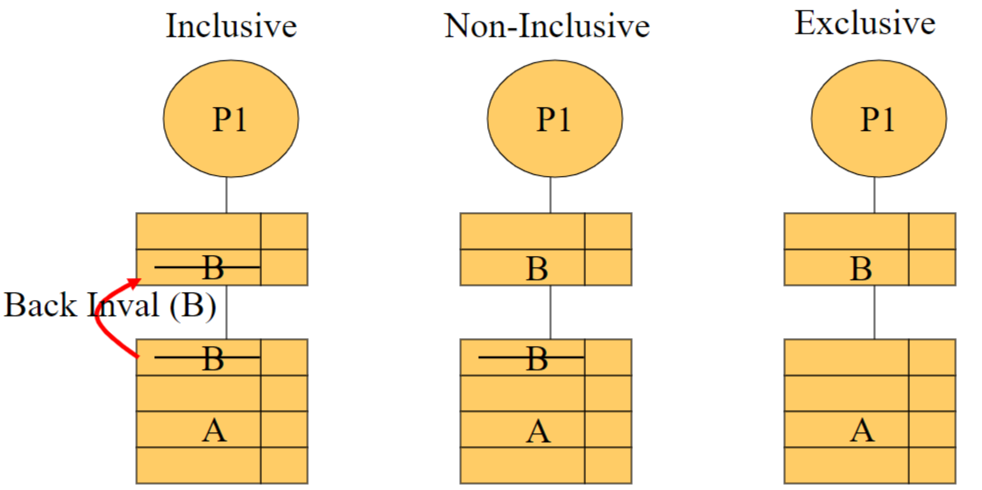
Fill at L1? Fill at L2?



Evict(A) from L1



Evict(B) from L2



4.3.1 Inclusive outer cache Pros/Cons

Pros:

- for most cases, external requests can be checked against the outer cache (if not in outer, cannot be in inner)
- reduces contention for cache tags at inner cache
- external snoop latency (and thus memory latency) is reduced significantly

Cons:

- space wasteful; forced data redundancy
- inflexible (power gating ways makes inner cache ineffective)

4.4 Cache Performance

4.4.1 Performance Metrics

- Average Access Time (AAT)
  - $AAT = L_1T + L_1MR \cdot L_2T + L_1MR \cdot L_2MR \cdot L_2MT$
  - L_nT = level n access time
  - L_nMR = level n miss rate
  - L_nMT = level n miss penalty

CPI and AAT

- CPI (cycles per instruction) measures impact of AAT on overall performance
- CPU time = $CPI \cdot IC \cdot AAT$
- $CPI = CPI_0 + MemF \cdot AAT$
  - CPI_0 = CPI with a perfect cache (0% miss rate and 0-cycle access time)
  - $MemF$ = fraction of instructions which are loads/stores
  - the latencies used (i.e. L_nMT) must be ones that are *not* overlapped with computation and must be amortized over concurrent memory accesses (i.e. divide by number of simultaneous memory accesses)

4.4.2 Examples

Suppose that we have a system with two levels of caches. The L1 cache has an access latency of 1 clock cycle, while the L2 cache has an access latency of 9 clock cycles. An L2 cache miss costs 100 clock cycles to service.

1. Calculate the average access time (AAT or AMAT) if an application suffers 20% L1 miss rate and 10% L2 miss rate.
2. Calculate AAT' if it is known that on average, 100% of L1 cache access latency is overlapped with computation, 50% of L2 cache access latency is overlapped with computation, and 0% L2 miss latency is overlapped with computation. Assume that on average, 2 memory references are serviced simultaneously for L2 cache accesses, and only 1.2 for L2 cache misses.
3. Calculate the CPI using AAT' if the perfect cache CPI is 0.5 and 20% of all instructions are memory references (load/store).

Answer:

1. use $AAT = L1T + L1MR \cdot L2T + L1MR \cdot L2MR \cdot L2MT$
   - $AAT = 1 + 0.2 \cdot 9 + 0.2 \cdot 0.1 \cdot 100 = 1 + 1.8 + 2 = 4.8$ clock cycles
2. New AAT'
   - $L_1T = 0$ (because it is completely overlapped with computation)
   - $L_2T = \frac{0.5 \cdot 9}{2} = 2.25$
   - $L_2MT = \frac{1.0 \cdot 100}{1.2} = 83.3$
   - $AAT' = 0 + 0.2 \cdot 2.25 + 0.2 \cdot 0.1 \cdot 83.3 = 0.45 + 1.67 = 2.12$
3. use $CPI = CPI_0 + MemF \cdot AAT'$
   - $CPI = 0.5 + 0.2 * 2.12 = 0.92$

4.5 Improving Cache Performance

4.5.1 Reduce miss rate

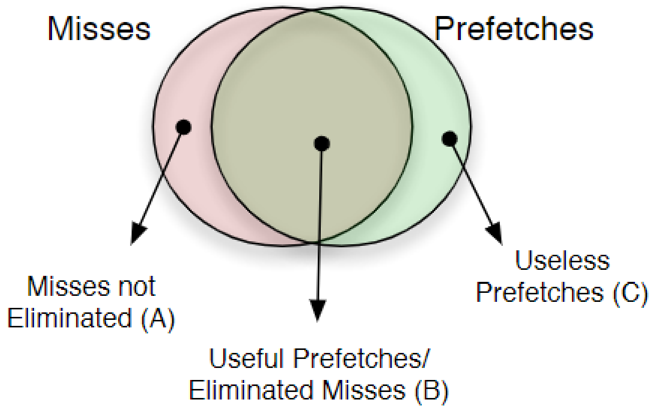
Types of cache misses:

- compulsory: misses required to bring blocks into the cache for the first time
- conflict: misses that occur due to insufficient cache associativity
- capacity: misses that occur due to finite cache size
- coherence: misses that occur due to invalidation by other processors
- system related: misses due to system activities such as system calls, interrupts, context switches, etc.

Parameters	Compulsory	Conflict	Capacity
larger cache size	unchanged	unchanged	reduced
larger block size	reduced	unclear	unclear
larger associativity	unchanged	reduced	unchanged

Reducing miss rate:

1. increase block size
   - (+) idea: exploit spatial locality
   - (-) overdoing it can lead to cache pollution from useless data
   - (-) also increases miss penalty (have to bring more data in each miss)
2. increase cache size
   - (+) larger caches hold more
   - (-) steals resources from other units
   - (-) diminishing returns: double size != double performance
   - (-) larger caches are slower to access
3. increase set associativity
   - (+) fully associative yields better performance than direct mapped
   - (-) slower (spend more time searching within a set)
   - (-) diminishing returns: 8-way set-associative often comparable (almost equivalent) in HR/MR to fully-associative
4. hashing for better set mapping within cache
   - useful if sets are not uniformly utilized (i.e. most of the time, addresses map to set index 0)
5. replacement or LRU insertion
   - LRU works well 80% of the time, but poorly when the working set is larger than cache
   - replacement: detect pathological cases and use random replacement
   - placement: detect pathological cases and insert at LRU
6. **Prefetching**
   - prefetch: get data in cache before it's needed/requested by the processor
   - important metrics:
     - coverage: fraction of misses prefetched
     - accuracy: fraction of prefetches that are useful
     - timeliness: implicitly part of accuracy, but also important to consider


Misses
Prefetches
Misses not Eliminated (A)
Useful Prefetches/ Eliminated Misses (B)
Useless Prefetches (C)

Coverage = B / (A+B)
Accuracy = B / (B+C)

   - next line prefetching:

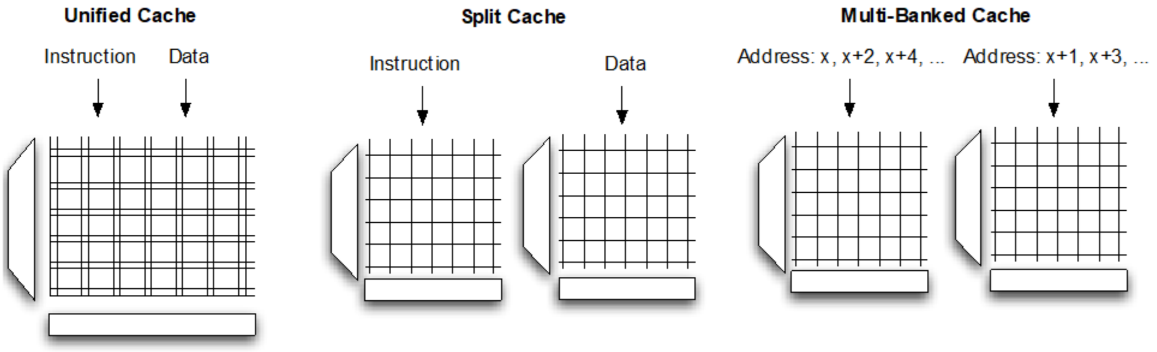
  - fetch missing/requested block *and* the next sequential block
  - works great for stream with high sequential locality i.e. instruction caches (Icaches)
  - uses unused memory bandwidth between misses (can hurt if there isn't much left-over bandwidth)
- stride prefetching
  - if memory is being accessed every n locations, then just prefetch block + n
  - for example, useful in a for loop that increments by n each iteration

7. other optimizations
   - loop interchange: increase temporal locality by exchanging inner and outer loops (i.e. 2d for loop should have i outer, j inner if accessing $arr[i][j]$)
   - loop fusion: two loops with identical sets of iterations can be combined into one loop (duplicated things in the first loop might be out of the cache by the time the second loop executes, so putting both in one loop reduces the chances of this)

4.6 Cache Performance Improvements cont.

4.6.1 Reduce hit time

- use small and simple caches: lower access time needed, and decreased power consumption)
- way prediction
  - goal is to keep hit time as low as possible (i.e. approach direct-mapped cache hit time)
  - use "Predicted Next Way" (PNW) bits for each block
  - when a block is accessed, read along its PNW bits
  - on the next access, only read the predicted way (which is just a single tag comparison)
  - if predicted correctly, you benefit from an extremely fast hit time; 90% acc. for 2-way, 80% for 4-way
  - if predicted incorrectly, you pay the penalty of accessing other ways and then also having to update the PNW bits
- pipelined cache access: just pipeline the stages for accessing in a cache (decode & activate row, tag compare, select bytes/words)
- split and multi-banked cache organization
  - in a unified L1 cache that holds instructions and data, each cycle it has to be accessed to supply instructions to the pipeline, and each cycle, several load/store instructions may access it. Thus, it needs to have many ports (expensive and slow)
  - solution: split into instruction cache + data cache
  - no longer contending for the same ports



4.6.2 Reduce miss penalty

- use multi-level caches
  - single level cache: too small, high miss rate; too large, high hit (access) time
  - multi-level cache: L1 (small but fast), L2 (slower but larger), L3 (even slower but even larger)

  - since it takes a long time to go down to main memory, just put a cache between L1 and main memory (L2 cache)

- use write buffers
  - for writes (write-through) or write backs (write-back), don't make CPU wait on the write to memory
  - use a write buffer: on read miss, check for match

- early restart/critical word first
  - early restart: as soon as requested word arrives, forward it to CPU; miss penalty time is now the time needed to fetch the requested word
  - critical word first: problem with early restart (what if requested word is in the middle of a block?); start fetching the block with the required (critical) word, and fill in the rest

- subblocking
  - problem: tags are overhead, and take up extra space
  - partial soln: large blocks reduce amount of tag storage (double blocksize means we halve the num. blocks, which also means we halve the number of tags); but large blocks increase miss penalty
  - complete soln: use large blocks, but also subdivide blocks into subblocks. Fetch only 1 subblock on a miss, keep valid bit for subblocks. Also better if other subblocks are prefetched in the background (aka combine subblocking with early restart/critical word first)

4.6.3 Reduce power consumption

- parallel vs. sequential cache access: parallel is faster, but sequential is more power efficient; use parallel for L1, and sequential for L2/L3

4.7 Virtual Memory

Program uses virtual address but translates (maps) it to a physical address at runtime.

4.7.1 Paging

- memory divided into chunks called pages, typically 4KB
- each page in program address space mapped into a page frame
- page has virtual address (VA), page frame has physical address (PA)
- OS keeps a page table that translates VA to PA; each entry is a page table entry (PTE)
- illusion of larger memory than physically available by swapping space in storage (page swapped in/out on demand)

4.7.2 Page fault

- page fault: exception raised because of illegal access (insufficient permission [write to read-only], page miss [page not in physical memory due to PTE being unmapped])
- page miss handling:
  - page miss incurs a page fault, triggers OS page fault handler
  - victim page selected and swapped out to swap space in disk
  - page read from disk and swapped into now-free page frame
  - page fault handler exits, and instruction that caused page fault re-executed

4.7.3 Accelerating page table access

- page table is large, so accessing it takes a while
- use a cache of a small number of PTEs that are recently used, called a Translation Lookaside Buffer (TLB)
- just like regular caches, a TLB can have/be:
  - split into instruction and data TLB
  - multi-level
  - replacement policies (LRU, PLRU, etc.)
  - associativity, blocksize, cache size is fixed to size of PTE

4.7.4 Page table size

- suppose we have a 48-bit address space, and page size is 4KB (hence 12-bit page offset)
- there are $48 - 12 = 36$ bits used for page address
- so there are $2^{36} = 64$ giga pages
- each PTE is 8 Bytes in size, so page table is $64\text{G} \cdot 8\text{B} = 512\text{GB}$
- each program has a page table, so we need $1000 \cdot 512\text{GB}$ for page tables normally; reduce this using hierarchical page tables

5 ILP Techniques: Pipelining

5.1 Pipeline Design

5.1.1 How to execute an instruction

5 stages:

1. Instruction fetch (IF)
   - instruction register (IR) = Mem[PC]
   - new PC = PC + 4
2. instruction decode/register fetch (ID)
   - A = Regs[$\text{IR}_{6..10}$]
   - B = Regs[$\text{IR}_{11..15}$]
   - Imm = sign-extend($\text{IR}_{16..31}$)
3. execute (EX)
   - memory reference: ALU output = A + Imm
   - reg/reg ALU operation: ALU output = A *op* B
   - reg/immediate ALU operation: ALU output = A *op* Imm
   - branch: ALU output = new PC + Imm; Cond = (A *op* 0)
4. memory access/branch completion (MEM)
   - memory reference
     - `load_mem_data = Mem[ALUOutput] // load`
     - `Mem[ALUOutput] = B // store`
   - branch
     - `if (cond) PC = ALUOutput; else PC = NPC`
5. write-back (WB)
   - reg-reg ALU operation: Regs[$\text{IR}_{16..20}$] = ALUOutput

- reg-immediate ALU operation: Regs[$\text{IR}_{11..15}$] = ALUOutput
- load instruction: Regs[$\text{IR}_{11..15}$] = `load_mem_data`

Pipeline speedups (no stalls) for a pipeline of n stages

$$\text{speedup} = \frac{\texttt{avg. exec time unpipelined}}{\texttt{avg. exec time pipelined}}$$

$$\text{speedup} = \frac{T_{unpipe}}{T_{unpipe}/n + T_{latch}\cdot(n-1)} = n \text{ (ideal case, } T_{latch} = 0\text{)}$$

- pipelining helps by keeping CT constant, while improving CPI
- or by keeping CPI constant, while improving CT
- but most of the time, pipelining does a bit of both improving CPI and CT

5.1.2 Pipeline hazards

Three kinds:

- data hazards: dependencies between instructions prevent their overlapped execution
- structural hazards: not enough hardware resources for all combinations of instructions (i.e. two multiply instructions need to be pipelined, but we only have one multiplier unit)
- control hazards: branches change the PC, which results in late code (if the wrong code is executed)

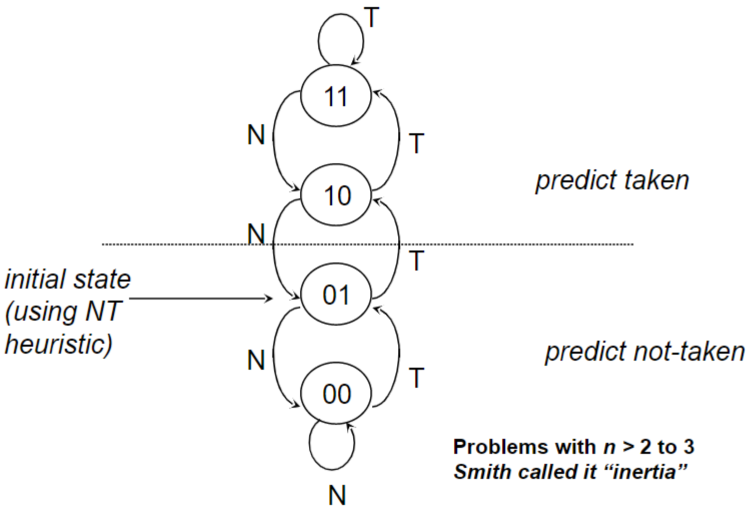
5.2 Branch Prediction 1

Idea: avoid branch stalls by predicting which way a branch will go (i.e. taken vs. not-taken)

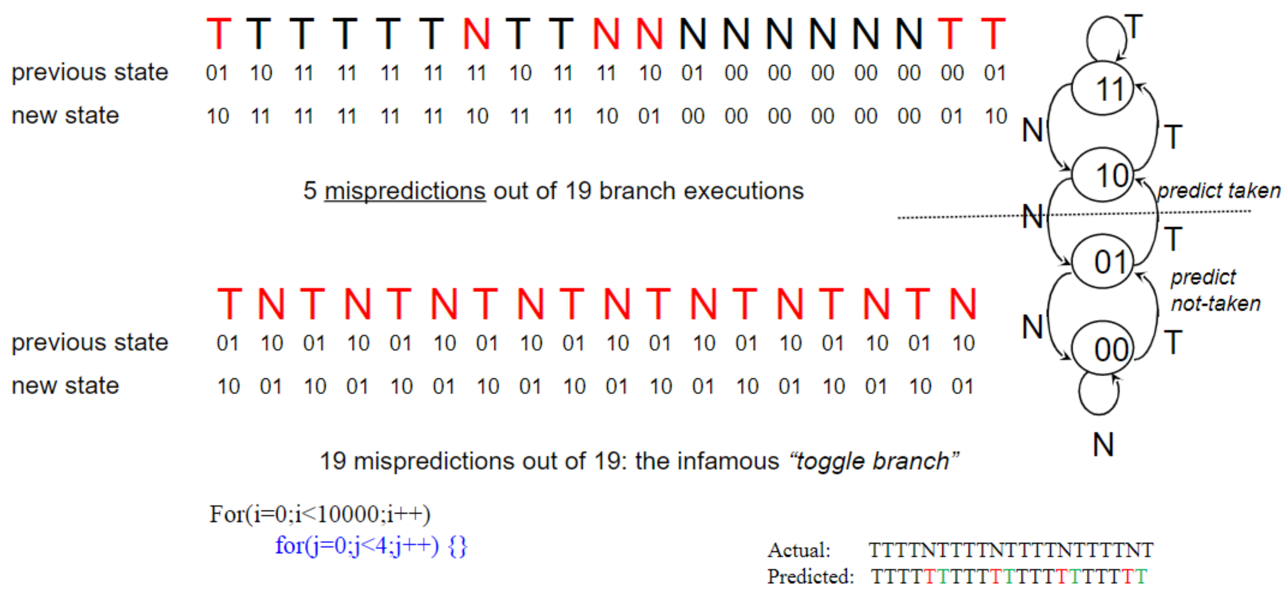
- perhaps use a prediction bit if branch taken, 0 if not taken. At instruction fetch, if prediction field is 1, predict taken, else not taken.
- problem: some branches may alternate or not do the same thing every time, so we need more sophistication

5.2.1 Smith n-bit counter predictor

Idea: replace prediction bit/field with an n-bit counter



Smith counter example (note the weakness on the alternating example):



T T T T T T N T T N N N N N N N N T T

previous state 01 10 11 11 11 11 11 10 11 11 10 01 00 00 00 00 00 00 01

new state 10 11 11 11 11 11 10 11 11 10 01 00 00 00 00 00 00 01 10

5 mispredictions out of 19 branch executions

T N T N T N T N T N T N T N T N T N

previous state 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10

new state 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01 10 01

19 mispredictions out of 19: the infamous *"toggle branch"*

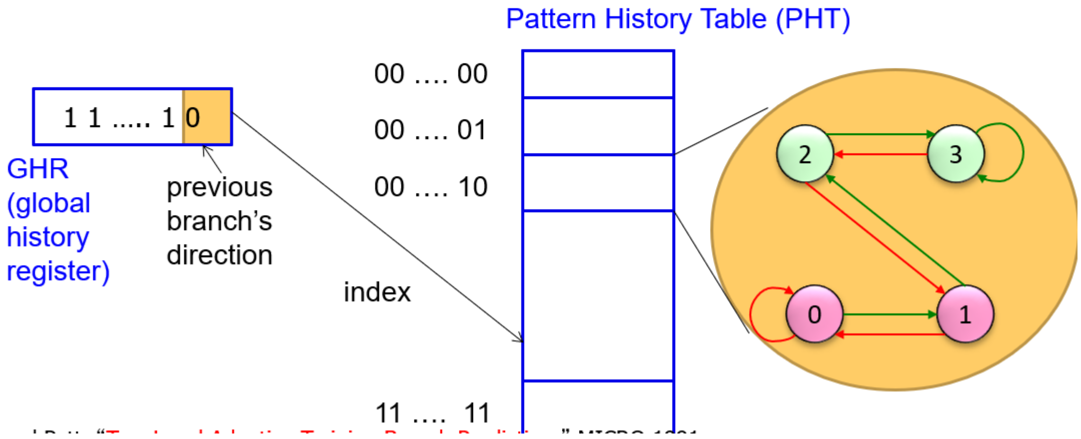
```
For(i=0;i<10000;i++)
        for(j=0;j<4;j++) {}
```

Actual: TTTTNTTTTNTTTTNTTTTNT
Predicted: TTTTTTTTTTTTTTTTTTTTT

5.2.2 Global Branch Correlation and Two-Level Global Branch Prediction

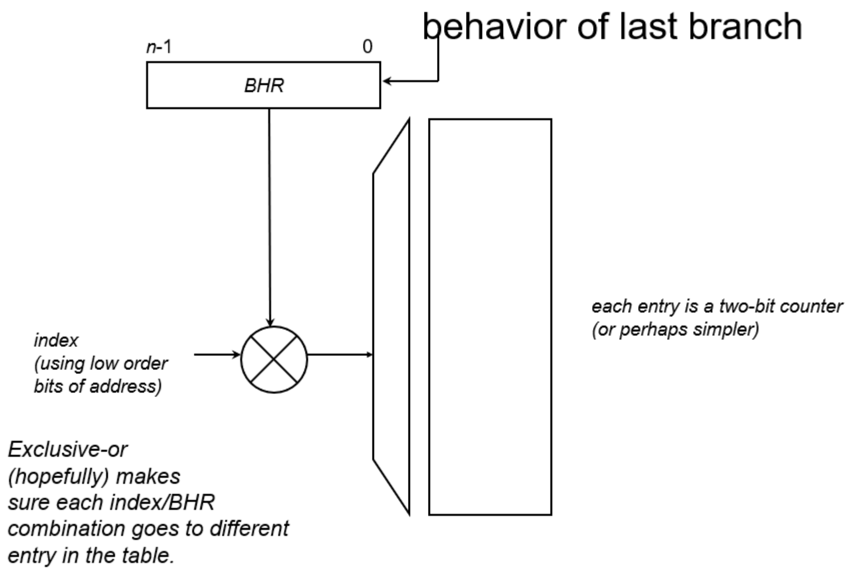
Associate branch outcomes with a global taken/not-taken history of all branches. Make a prediction based on the outcome of the branch the last time the same global branch history was encountered.

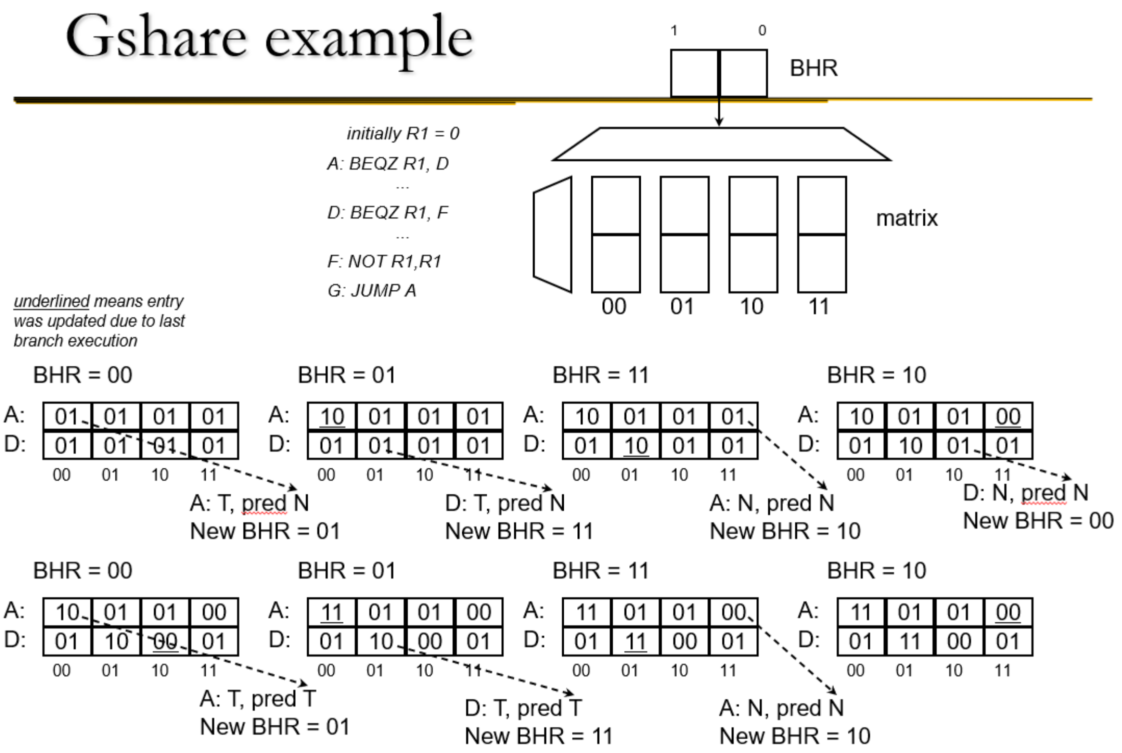
- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



5.2.3 Gshare predictor

Problem with two-level global branch predictor was that prediction was only based on the global history of all branches, independent of individual branches. Gshare adds more context by also considering individual branches (using both PC and global branch history register to index into the pattern history table).





Gshare example

1 0

BHR

matrix

00 01 10 11

initially R1 = 0
A: BEQZ R1, D
...
D: BEQZ R1, F
...
F: NOT R1,R1
G: JUMP A

underlined means entry was updated due to last branch execution

BHR = 00

	00	01	10	11
A:	01	01	01	01
D:	01	01	01	01

A: T, pred N
New BHR = 01

BHR = 01

	00	01	10	11
A:	10	01	01	01
D:	01	01	01	01

D: T, pred N
New BHR = 11

BHR = 11

	00	01	10	11
A:	10	01	01	01
D:	01	10	01	01

A: N, pred N
New BHR = 10

BHR = 10

	00	01	10	11
A:	10	01	01	00
D:	01	10	01	01

D: N, pred N
New BHR = 00

BHR = 00

	00	01	10	11
A:	10	01	01	00
D:	01	10	00	01

A: T, pred T
New BHR = 01

BHR = 01

	00	01	10	11
A:	11	01	01	00
D:	01	10	00	01

D: T, pred T
New BHR = 11

BHR = 11

	00	01	10	11
A:	11	01	01	00
D:	01	11	00	01

A: N, pred N
New BHR = 10

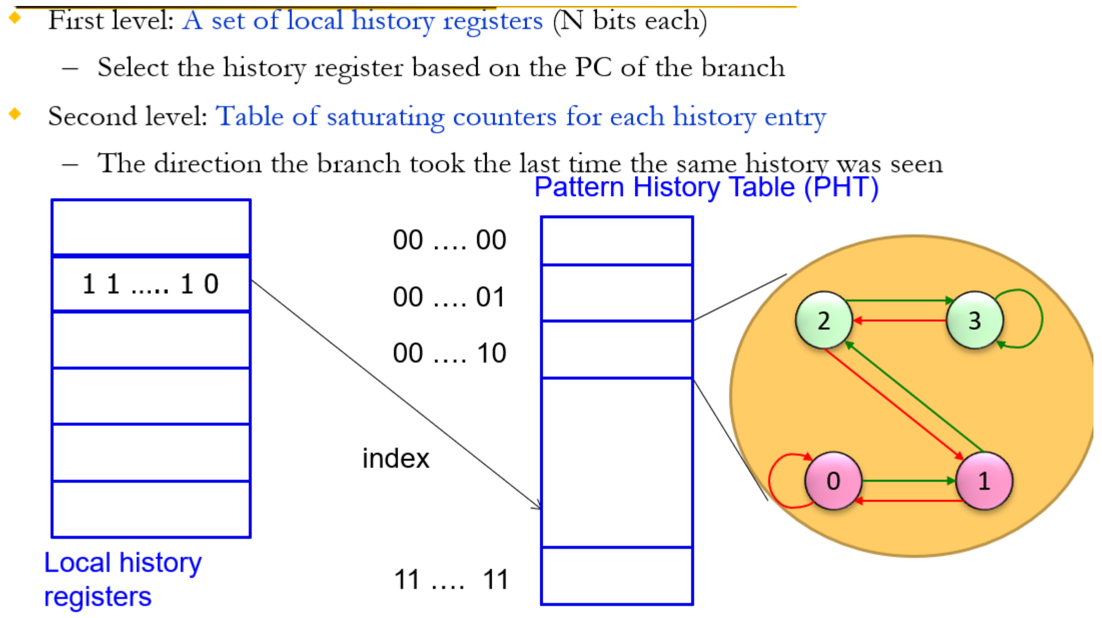
BHR = 10

	00	01	10	11
A:	11	01	01	00
D:	01	11	00	01

5.3 Branch Prediction 2

5.3.1 Two-Level Local Branch History

Improving on Gshare: Local history with the Yeh/Patt Two-Level Local Branch predictor



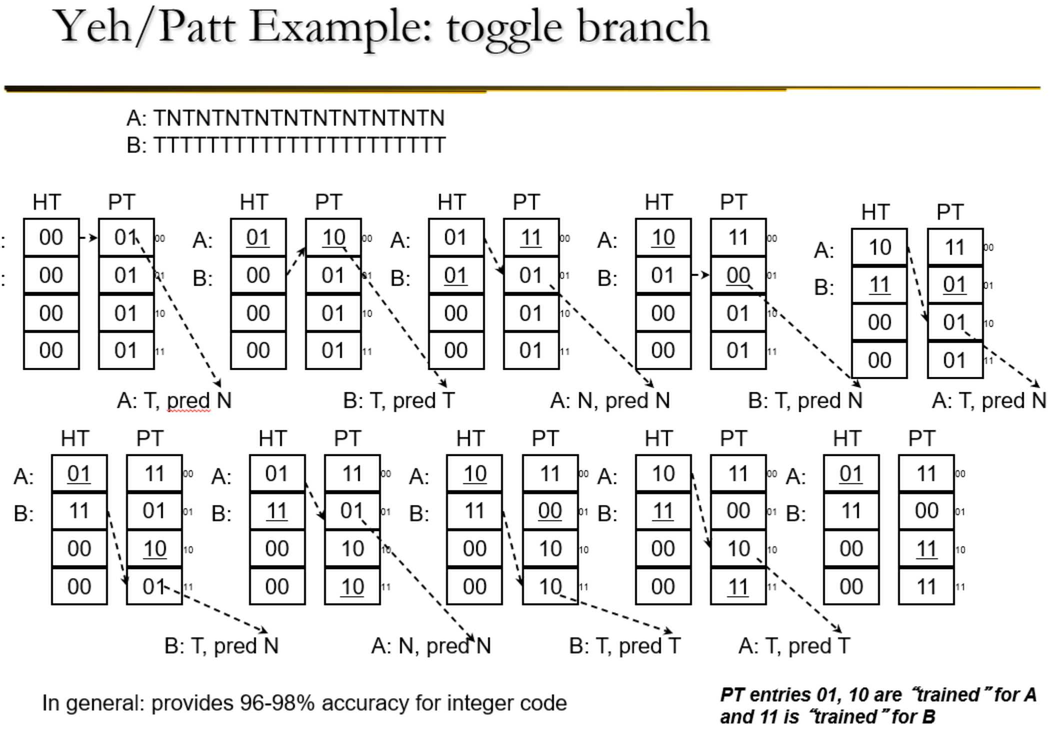
- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Pattern History Table (PHT)

1 1 ..... 1 0

00 00
00 01
00 10
index
11 11

Local history registers



Yeh/Patt Example: toggle branch

A: TNTNTNTNTNTNTNTNTNTN
B: TTTTTTTTTTTTTTTTTTTTTTT

	HT	PT	
A:	00	01	00
B:	00	01	01
	00	01	10
	00	01	11

A: T, pred N

	HT	PT	
A:	01	10	00
B:	00	01	01
	00	01	10
	00	01	11

B: T, pred T

	HT	PT	
A:	01	11	00
B:	01	01	01
	00	01	10
	00	01	11

A: N, pred N

	HT	PT	
A:	10	11	00
B:	01	00	01
	00	01	10
	00	01	11

B: T, pred N

	HT	PT	
A:	10	11	00
B:	11	01	01
	00	01	10
	00	01	11

A: T, pred N

	HT	PT	
A:	01	11	00
B:	11	01	01
	00	10	10
	00	01	11

B: T, pred N

	HT	PT	
A:	01	11	00
B:	11	01	01
	00	10	10
	00	10	11

A: N, pred N

	HT	PT	
A:	10	11	00
B:	11	00	01
	00	10	10
	00	10	11

B: T, pred T

	HT	PT	
A:	10	11	00
B:	11	00	01
	00	10	10
	00	11	11

A: T, pred T

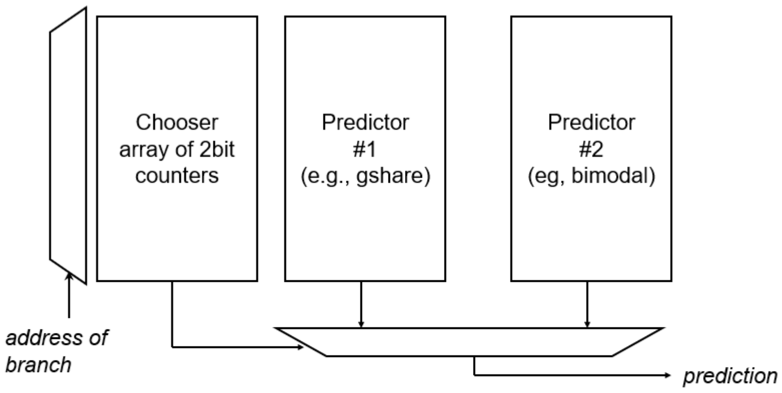
	HT	PT	
A:	01	11	00
B:	11	00	01
	00	11	10
	00	11	11

In general: provides 96-98% accuracy for integer code

PT entries 01, 10 are "trained" for A and 11 is "trained" for B

5.3.2 Hybrid/Tournament predictors

There is no one-size-fits-all solution to branch prediction; some are just better for some programs while worse in others. So, the idea is to use a combination of them, pick the one that is correct more often, and over time lean/have a bias toward that one.


Chooser array of 2bit counters
Predictor #1 (e.g., gshare)
Predictor #2 (eg, bimodal)
address of branch
prediction

- Both predictors supply a prediction-- pipeline uses only one
- Chooser updated based on which predictor was correct
  - Increment chooser counter if #1 was correct, decrement if #2 was correct

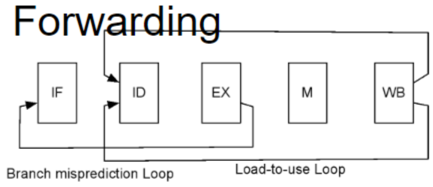
5.4 Dependence Hazards and Pipeline Performance

5.4.1 Types of dependencies

- True dependence
  - `ADD R1, R2, R3`
  - `SUB R4, R5, R1`
  - may cause **RAW** hazards
- Anti-dependence
  - `ADD R3, R2, R1`
  - `SUB R1, R4, R5`
  - may cause **WAR** hazards
  - due to reuse, can be removed by just using another register
- Output-dependence
  - `ADD R1, R2, R3`
  - `SUB R1, R4, R5`
  - may cause **WAW** hazards
  - due to reuse, can be removed by just using another register

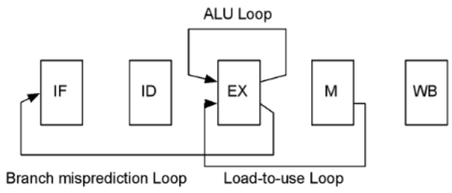
5.4.2 Pipeline performance and loop analysis

- Without Data Forwarding



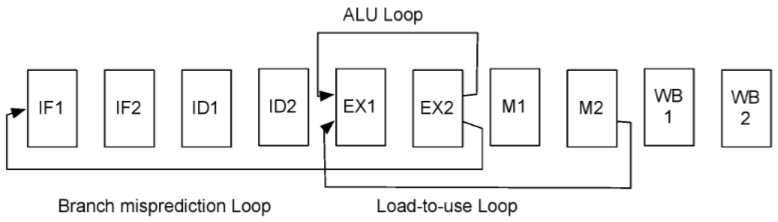
  - ALU Loop size = 4
  - Load-to-use Loop = 4
  - Branch Loop = 3

- With Data Forwarding



  - ALU Loop size = 1
  - Load-to-use Loop = 2
  - Branch Loop = 3

Case 1: Deep Pipeline

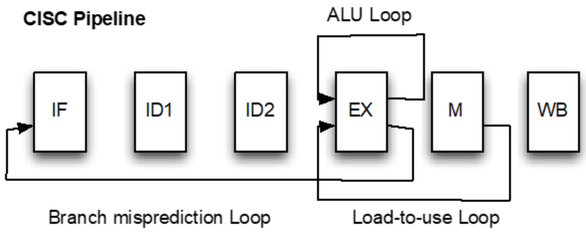


- ALU Loop size = 2 (was 1)
- Load-to-use Loop = 4 (was 2)
- Branch Loop = 6 (was 3)

- Thus, deepening the pipeline also increases the critical loop size
  - Benefit primarily comes when instructions do not have dependences, or when branches are easy to predict
  - Diminishing return from deepening the pipeline

Case 2: RISC vs. CISC Pipeline

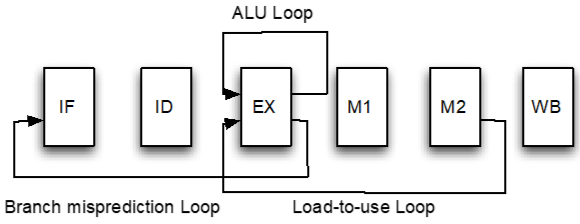
- CISC
  - Assume CISC instruction decoded and broken into small RISC-like instructions (e.g. Macro Op -> Micro Op)
  - Question: is there a cost for doing this vs. RISC?



  - ALU Loop and Load-to-use Loop have the same size as RISC
  - But Branch Loop = 4 (vs. 3 in RISC) => branch misprediction is costlier
- Thus, everything else the same, must compensate with a better branch predictor

Case 3: Increasing the L1Cache Size

- Larger L1 Data Cache Size
  - Suppose doubling the size adds 1 additional cycle (one extra “M2” stage)
  - Question: what is the performance cost for doing this?



  - Load-to-use Loop is now 3 cycles (was 2 cycles)
- Thus, worse performance for instructions dependent on the loads
  - Can compensate by scheduling the loads early
  - May also avoid increasing L1 cache size further, back it up with a large L2 cache instead