Computer Architecture

• Instruction Set Architecture (Ch 2)

- Classification: By Internal Storage
  - Stack: operands implicit
  - Accumulator: one operand implicit
  - Registers: both operands explicit
    - Register-Mem: can access memory as part of any instruction. **Adv:** data accessed w/o a sepearate load instruction. Instruction format easy to encode, good density. **Disadv:** operands not equiv(source operand in binary op destroyed). Register number and mem address in each instruction restricts number of registers. CPI vary by instruction location
    - Load-Store/Register-Register: can access memory as part of load/store only. more popular b/c registers faster than memory, and registers more efficient for compiler use. **Adv:** simple, fixed length instructions. Simple code generation, Instructions use similar # of cycles. **Disadv:** higher instruction count than register-mem. More instructions&lower instruction density=larger programs.
    - Memory-Memory: all operands in memory. most compact. doesnt waste registers for temporaries disadv: large variation in instruction size, esp. for 3 operand instructions. large variation in work per instruction. memory bottleneck
- Memory Addressing:
- Stack architecture
  - Operands implicitly on top of a stack.
- Accumulator architecture
  - One operand is implicitly an accumulator
- General-purpose register architecture (both operands explicit)
  - Register-memory architectures
    - One operand can be memory.
    - Adding reg-mem only better than reg-reg if load instructions have no offset
    - Could be worse than reg-reg if only a load instruction
    - **Adv:**
      - data accessed w/o a sepearate load instruction.
      - Instruction format easy to encode, good density.
    - **Disadv:**
      - operands not equiv(source operand in binary op destroyed).
      - Register number and mem address in each instruction restricts number of registers.
      - CPI vary by instruction location
  - Register-Register (Load-store) architectures
    - All operands are registers (except for load/store)
    - **Adv:**
      - simple, fixed length instructions.
      - Simple code generation,
      - Instructions use similar # of cycles.
    - **Disadv:**
      - higher instruction count than

Problem solving:

Adding reg-mem to a reg-reg pipeline

Original Pipeline	Modified pipeline	Comments
LD R9, 16(R1)	ADDUI R2, R1, #16 LD R9, 0(R2)	Required an additional instr.
LD R9, 8(R1) ADD R7, R6, R8	ADDUI R2, R1, #8 ADD R7, R8, 0 (R2)	No gain , no loss. See [6.e]
LD R6, 0(R1) ADD R7, R6, R8	ADD R7, R8, 0 (R1)	Two instructions merged.

Example	Original Pipeline	Modified pipeline
LD R6, 0(R1) ADD R7, R6, R8	1 Cycle stall	No Stalls
ADDUI R6, R5, #8 LD R7, 0(R6)	No Stall	1 Cycle Stall

- Execution Pipeline (Appendix A)

1. **IF**: Instruction Fetch

- Send the program counter to memory and fetch the current instruction from memory
- Update the PC to the next sequential PC by adding 4 to the PC

2. **ID**: Instruction decode/ register fetch cycle

- Decode the instruction and read the registers corresponding to register source specifiers from the register file
- Do the equality test on the registers as they are read, for a possible branch
- Sign extend the offset field of the instruction in case it is needed
- Compute the possible branch target address by adding the sign-extended offset to the incremenented PC
- Branch could be completed at the end of this stage (store the branch-target address into the PC)

3. **EX**: Execution/ Effective address cycle

- ALU operates of the operands prepared in the ID cycle. One of three functions:
    - Memory reference: ALU adds the base register and the offset to form the effective address
    - Register-Register ALU instruction: The ALU performs the operation specified by the ALU opcode on the values read from the register file
    - Register-Immediate ALU instruction: The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate
    - In load-store, the effective address and execution cycles can be combined into one b/c no instruction needs to calculate a data address and perform an operation on the data

4. **MEM**: Memory Access/branch completion

- If the instruction is aload, memory does a read using the effective address from EX.
- If it is a store, memory writes the data from the second register read from the register file, using the effective address

5. **WB**: Write-back cycle

- Register-Register ALU instruction or Load instruction: write the result to the register file -- coming from the memory system (a load) or the ALU (a store)

- Clock Period
  - τ = Max { time delay of a stage }1k + other delay (e.g., skew, latch delay)
- Frequency
  - $f = 1/\tau$
  - Reciprocal of the clock period
- Speedup
  - k stage pipeline, n instructions
  - $S_k = \frac{n.k}{k+(n-1)}$ → k when n >> k.
- Speedup (w/Stalls)
  - Avg. inst. time (unpipelined) / Avg. inst. time (pipelined)
  - CPI unpipelined / CPI pipelined
  - Pipeline Depth / (1 + Pipeline stall cycles per instruction)
  - Pipeline Depth / (1 + Branch frequency x Branch penalty)

- Efficiency
  - $\eta = S_k / k$
  - Ratio of its actual speedup to the ideal speedup
- Throughput
  -
  - Number of instructions that can be completed per cycle.

- ## Pipeline Hazards
  - Let *i* be an earlier instruction, *j* a later one.
  - **RAW** (read after write)
    - *j* tries to read a value before *i* writes it
  - **WAW** (write after write)
    - *i* and *j* write to same place, but in the wrong order.
    - Only occurs if >1 pipeline stage can write (in-order)
  - **WAR** (write after read)
    - *j* writes a new value to a location before *i* has read the old one.
    - Only occurs if writes can happen before reads in pipeline (in-order).

Exceptions

- **Types**
  - Synchronous vs. asynchronous
    - Event synchronized with program execution?
  - User requested vs. coerced
    - Event caused intentionally by user program?
  - User maskable (can be disabled) or not

        - Can event be disabled?
    - Within instructions or between instructions
        - Does event prevent instruction from completing?
    - Resume vs terminate
        - Does the program continue from where it left off after exception is handled, or does it stop?

- **Restartable Exceptions:**
    - Requirements:
        - Exception may occur within instruction.
        - Program must continue after exception is handled.
    - Examples:
        - Virtual memory page fault.
    - Difficult because:
        - Pipeline state must be saved.
    - One approach, for easy cases:
        - 1. Force a trap inst. into pipeline on next IF.
        - 2. Clear pipeline behind faulting instruction.
        - 3. Exception handler saves PC of faulting instr.

- Memory Hierarchy (Ch5)

Principle of Locality:

- Temporal locality - the same location likely to be accessed again soon
- Spatial locality - nearby locations likely to be accessed again soon

Level Behavior described by:

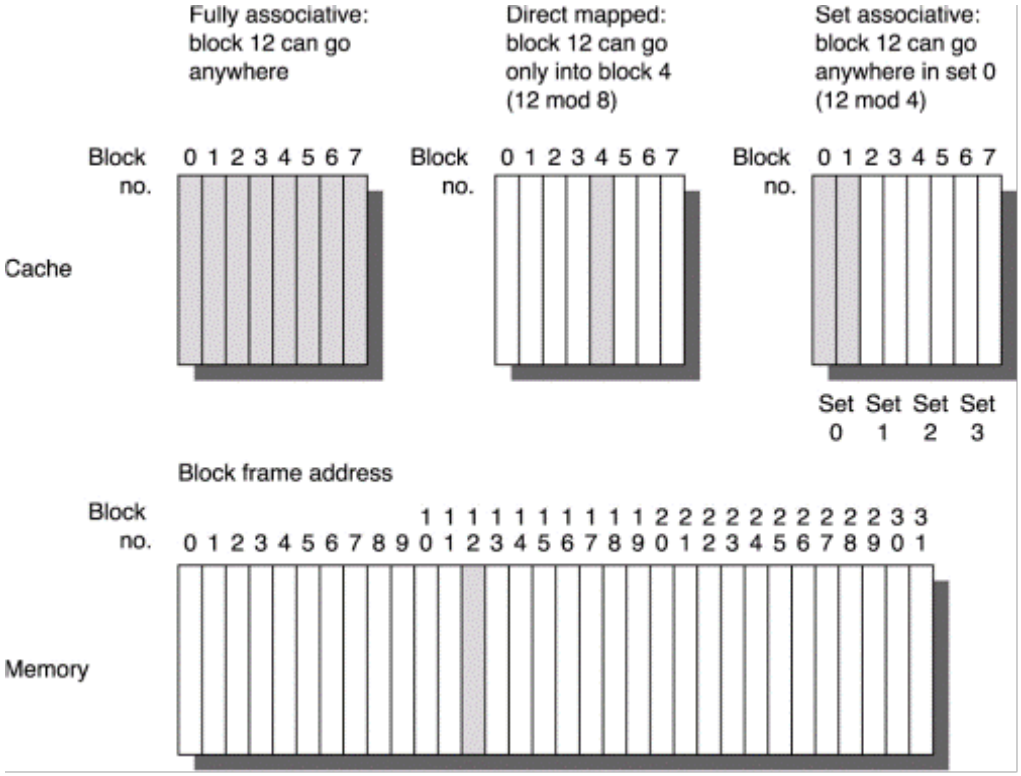
- Block Placement:
    - Where could a new block be placed in the level?
- Block Identification:
    - How is a block found if it is in the level?
- Block Replacement:
    - Which existing block should be replaced if necessary?
- Write Strategy:

CPU Execution time = (CPU clock cycles + Memory stall cycles) · Clock cycle time

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \cdot \text{Miss penalty} \\ &= IC \cdot \frac{Misses}{Instruction} \cdot \text{Miss penalty} \\ &= IC \cdot \frac{Memory\_access}{Instruction} \cdot \text{Miss Rate} \cdot \text{Miss penalty}\end{aligned}$$

Block placement:

- *Direct mapped*. (Block address) MOD (Nr of blocks in cache)
- *Fully associative*. Block can be placed anywhere in the cache
- *N-way set associative*. Block is first mapped to set and can then be placed anywhere in the set. Frame set# = (Block address) MOD *n*



Cache Size Equations:

- Simple equation for the size of a cache:
  - (Cache size) = (Block size) × (Number of sets) × (Set Associativity)
- Can relate to the size of various address fields:
  - (Block size) = 2^(# of offset bits)
  - (Number of sets) = 2^(# of index bits)
  - (# of tag bits) = (# of memory address bits) – (# of index bits) – (# of offset bits)

Block replacement:

- *Random*, randomly select a block to replace
- *Least-recently used (LRU)*, dispose the block that is used the latest
- *First in, first out (FIFO)*, LRU can be complicated and expensive, so instead we throw out the oldest block.

Block address		Block Offset
Tag	Index	

- *Block offset* – Selects the desired data from block
- *Index field* – Selects the set
- *Tag field* – Used to compare for a hit in the selected set, we only need to do this because of the following:
  - The offset should not be used in the comparison., since the entire block is present or not

- Checking the index is redundant, since it was used to select the set to be checked. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

Writing:

- *Write through* – Information is written to both block in cache and in lower-level memory
- *Write back* – The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
- *Write allocate* – The block is allocated on a write miss. Write missed are same as read misses.
- *No-write allocates* – Block is only modified in lower-level memory.

Average memory access time = Hit time + Miss rate · Miss penalty

In out of order execution we define a stall of miss penalty as nonoverlapped latency:

$$\frac{Mem\_stall\_cycles}{Instruction} = \frac{Misses}{Instruction} \times (Total\ miss\ latency - Overlapped\ miss\ latency)$$

- *Length of memory latency* – What to consider as the start and end of a memory operation in an out-of-order processor
- *Length of latency overlap* – What is the start of overlap with the processor. (When is a memory operation stalling the processor)

Summary of all the formulas:

1. $2^{index} = \frac{CacheSize}{BlockSize \times SetAssociativity}$
2. $CPU\ execution\ time = (CPU\ clock\ cycles + Memory\ stall\ cycles) \times Clock\ cycle\ Time$
3. $Memory\ stall\ cycles = Number\ of\ misses \times Miss\ penalty$
4. $Memory\ stall\ cycles = IC \times \frac{Misses}{Instruction} \times Miss\ penalty$
5. $\frac{Misses}{Instruction} = Miss\ rate \times \frac{MemoryAccesses}{Instruction}$
6. $Average\ memory\ access\ time = Hit\ time + Miss\ rate \times \text{Miss penalty}$
7. $CPU\ execution\ time = IC \times \left(CPI_{execution} + \frac{Memory\_stall\_clock\_cycles}{Instruxtion} \right) \times Clock\_cycle\_tine$
8. $CPU\ execution\ time = IC \times \left(CPI_{execution} + \frac{Misses}{Instruxtion} \times MissPenalty \right) \times Clock\_cycle\_tine$
9. $CPU\ execution\ time = IC \times \left(CPI_{execution} + MissRate \times \frac{MemoryAccesses}{Instruxtion} \times MissPenalty \right) \times Clock\_cycle\_tine$
10. $\frac{MemoryStallCycles}{Instruction} = \frac{Misses}{Instruction} \times (TotalMissLatency - OverlappedMissLatency)$

There are basically four strategies to speed up caches/memory:

1. **Reducing the miss penalty**: Multi level caches, critical word first and early restart, read miss before write miss, merging write buffers, victim caches
2. **Reducing the miss rate**: Larger block size, larger cache size, higher associativity, way prediction and pseudoassociativity, compiler optimizations.
3. **Reducing miss penalty or miss rate using parallelism**: Non-blocking caches, hardware prefetching, compiler prefetching
4. **Reducing time to hit in cache**: Small and simple caches, avoiding address translation, pipelined cache access, trace caches.

Reducing Miss Penalty

Mutli level caches: Just create more levels of caches:

- Average memory access time = Hit time (L1) + Miss rate (L1) x Miss Penalty (L1)
- Miss penalty (L1) = Hit time (L2) + Miss rate (L2) x Miss Penalty (L2)

- *"Local miss rate"*
  - The miss rate of one hierarchy level by itself.
  - # of misses at that level / # accesses to that level

    - e.g. Miss rate(L1), Miss rate(L2)
- *"Global miss rate"*
    - The miss rate of a whole group of hierarchy levels
    - # of accesses coming out of that group (to lower levels) / # of accesses to that group
    - Generally this is the product of the miss rates at each level in the group.
    - Global L2 Miss rate = Miss rate(L1) × Local Miss rate(L2)

Critical Word First & Early restart:

- *Critical word first* – Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling in the rest of the words in the block. Also called *wrapped* fetch and *requested word first.*
- *Early restart* – Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Generally these techniques only benefit designs with large cache blocks.

Giving Priority to Read Misses over Writes

- We server reads before writes have been completed.

Merging Write buffer

- If the buffer contains an address of a valid write buffer entry, then merge the new data.

Victim Caches

- Cache contains blocks that are discarded from a cache because of a miss. Victims are checked on a miss to see if they have the desired data before going to the next lower level of memory. If found, victim and cache block are swapped. Victim caches of size one to five are effective. Especially for small direct mapped data caches.

Reducing Miss Rate

There are three categories of misses:

1. *Compulsory* – The very first access to a block *cannot* be in the cahce, so the block must be brought into the cache. These are also called *cold-start misses* or *first-reference misses.*
2. *Capacity* – If the cache cannot contain all the blocks need during execution of a program, capacity misses (in addition o compulsory misses) will occur because of blocks being discarded and later retrieved.
3. *Conflict* – If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses.* The idea is that hits in a fully associative cache that become misses in an *n*-way set associative cache are due to more than *n* requests on some popular sets.

Here are four divisions of conflict misses and how they are calculated:

- *Eight-way* – Conflict misses due to going from fully associative (no conflicts) to eight-way associative
- *Four-way* – Conflict misses due to going from eight-way to four-way associative

- *Two-way* – Conflict misses due to going from four-way to two-way associative
- *One-way* – Conflict misses due to going from two-way to one-way associative (directly mapped)

Larger block size:

- Keep cache size & associativity constant
  - Fewer sets.
- Reduces compulsory misses
  - Due to spatial locality
  - More accesses are to a pre-fetched block
- Increases capacity misses
  - More unused locations pulled into cache
- May increase conflict misses (slightly)
  - Fewer sets may mean more blocks utilized per set
  - Depends on pattern of addresses accessed
- Increases miss penalty - longer block transfers

Larger Caches:

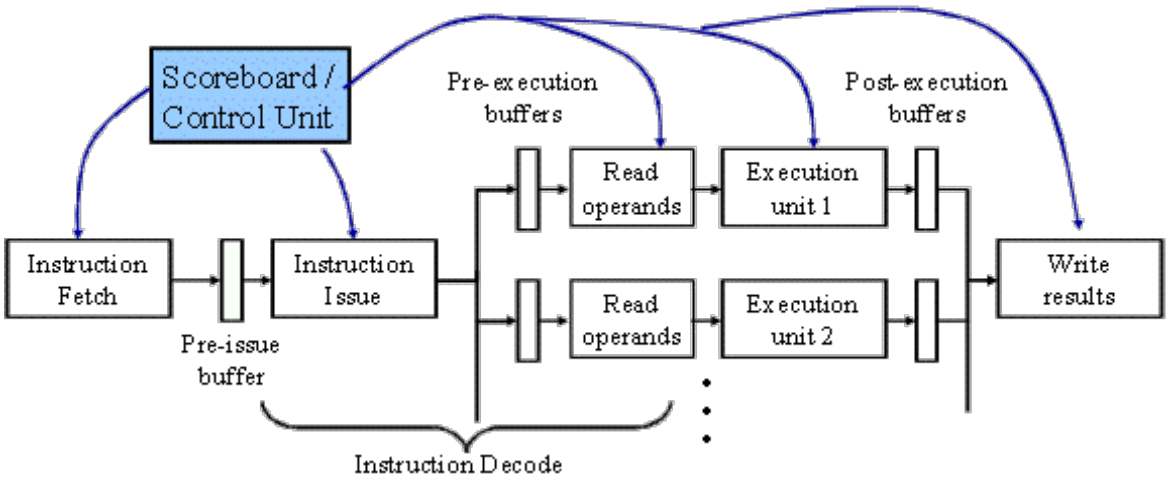
- Keep block size, set size, *etc.* constant
- No effect on compulsory misses.
  - Block still won't be there on its 1st access!
- Reduces capacity misses
  - More capacity!
- Reduces conflict misses (in general)
  - Working blocks spread out over more frame sets
  - Fewer blocks map to a set on average
  - Less chance that the number of active blocks that map to a given set exceeds the set size.
- But, increases hit time! (And cost.)

Higher associativity:

- Keep cache size & block size constant
  - Decreasing the number of sets
- No effect on compulsory misses
- No effect on capacity misses
  - By definition, these are misses that would happen anyway in fully-associative
- Decreases conflict misses
  - Blocks in active set may not be evicted early
    - for set size smaller than capacity
- Can increase hit time (slightly)
  - Direct-mapped is fastest
  - *n*-way associative lookup a bit slower for larger *n*

- I/O Subsystem
- Advanced Microarchitecture Issues:
  - Dynamic instruction scheduling(Ch 3.3)

Basic Scoreboard:



Instruction Issue Stage:

- Receive newly-fetched instruction
- Decode binary instruction format
- Check for structural hazards:
  - Instruction needs execution unit currently in use, whose initiation interval hasn't passed?
- Check for **WAW** hazards:
  - Instruction wants to write to a register that an active instruction (issued, but not yet finished) wants to write to?
    - Bad if they finish out-of-order!
- Stall all current (& future) instruction issuing, until none of these hazards remain.
- Issue instructions (in-order) to the appropriate execution units & track status on scoreboard.

Read Operands:

- Receive instruction issued to functional unit.
- Check for **RAW** hazards: Are all source operands available yet?
  - If **no**: Hold instruction in a pre-execution buffer.
    - If buffer has only 1 entry, this and all not-yet-issued instructions using this functional unit must wait.
  - If **yes**: Read operands from register file, & start instruction down the execution unit's pipeline.

Exection Stage

- Once operands are received, begin execution of the instruction in the execution unit.
- Execution may take multiple cycles.
- When result is ready, notify scoreboard of instruction completion.

Write Result Stage:

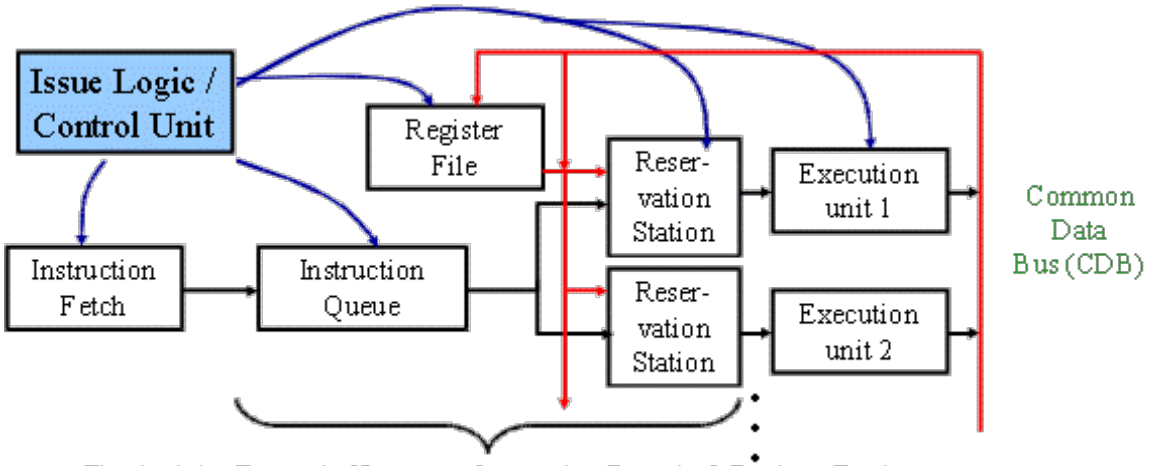
- Receive completed instruction & its result from execution unit.
- Check for **WAR** hazards:
  - Does any previously-issued instruction that has not yet read its operands depend on the old value we are about to overwrite? (Does it anti-depend on us?)

- While **yes**:
  Stall instruction in a post-execution buffer.
  - When **no**:
    Write instruction result to register file.

Tomasulo's Algorithm:

Components Of Tomasulo Algorithm:

- Reservation stations (RSs)
- Buffer the operands to pending instructions while they are waiting to enter the execution units.
- Effectively provides extra, non-programmer-visible “renaming” registers, dynamically avoids WAW/WAR hzds.
- Issue logic
- Redirects (renames) instructions’ register outputs to reservation-station slots.
- Results go directly to RSs rather than thru register file.
- Distributed hazard detection
- Handled separately by each functional unit
- Load & store buffers



The Activity Formerly Known as Instruction Decode & Register Fetch

Major Techniques:

- Dynamic Scheduling
- Register Renaming
- Dynamic Memory Disambiguation

Steps:

1. Issue

Get instruction from FP operation queue
If a slot in appropriate RS (or load-store buffer) is available, send instruction there; else stall it (structural hazard).
Send operand values to RS if already available, otherwise, just note the names of the operands in the RS
Rename registers

2. Execute.

While operands not yet available, monitor CDB for them.
When all operands are in RS, begin executing instruction.

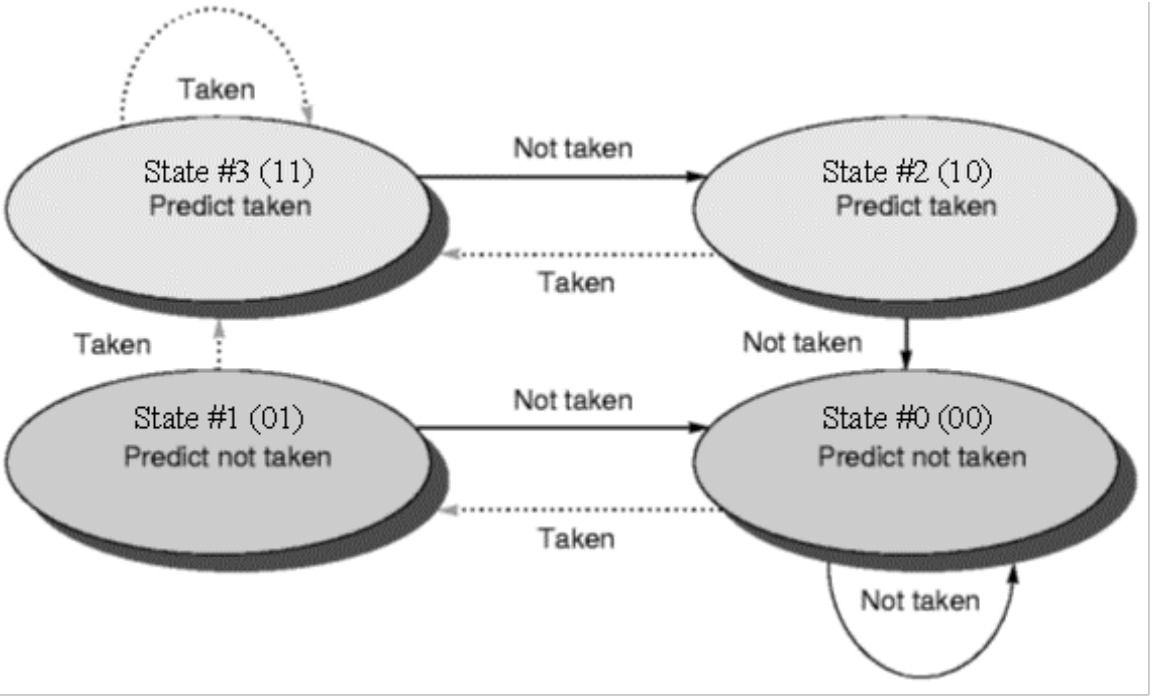
3. Write result.

When result available & CDB is free, write result to CDB, then to registers & RS/store slots for receiving instructions.

- Branch prediction (Ch 4.2, 3.4)

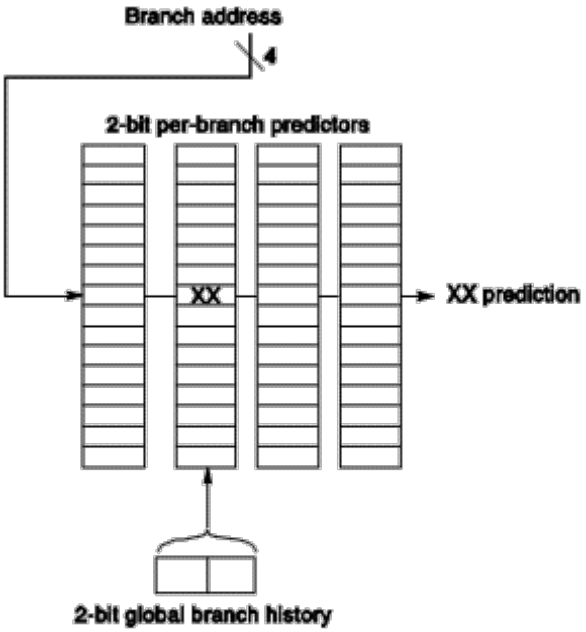
Basic Prediction

The idea is to resolve the outcome of a branch as early as possible, preventing control dependences from causing stalls. Effectiveness depends on accuracy and cost of a branch when a prediction is correct/incorrect.
Branch history table: Small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that tells whether or not a branch was taken.
n-bit prediction work by using an *n*-bit saturating counter for each entry in the prediction buffer. When the counter is greater than or equal to one half of its max value (), the branch is predicted as taken. The counter is incremented on a taken branch and decremented otherwise.



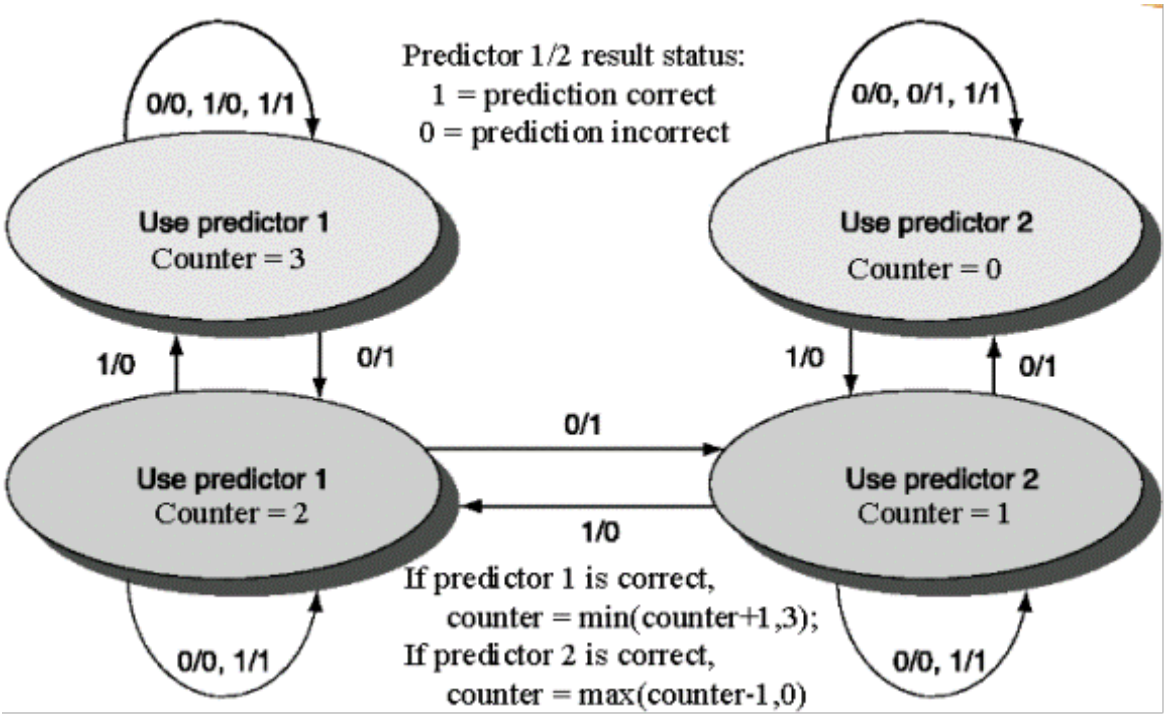
Correlating Branch Prediction

A correlating branch predictor is a branch predictor that uses the behavior of other branches to make a prediction. An *(m,n)* predictor uses the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor for a single branch.
The number of bits in an *(m,n)* predictor is 2^m x n x *number of prediction entries selected by branch address*



Tournament Predictors

Tournament predictors combine both ideas mentioned above. They use a predictor to select the type of branch predictor to use for the current branch. For example we could predict whether or not to use a correlation branch predictor or a normal one for this branch.



- Look-up Free caches
  - Known as *lookup-free cache*, *hit under miss*
  - While a miss is being processed,
    - Allow *other* cache lookups to continue anyway
  - Useful in dynamically scheduled CPUs
    - Other instructions may be in the load queue
  - Reduces *effective* miss penalty
    - Useful CPU work fills the miss penalty "delay slot"
  - *hit under multiple miss*, *miss under miss*:
    - Extend technique to allow multiple misses to be queued up, while still processing new hits

- Instruction-Level Parallelism (Ch 3)

Data Dependences

We have a data dependency when the following holds:

- Instruction *a* produces a result that may be used by instruction *b*, or
- Instruction *b* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *a*.

The second one simple states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions.

Name Dependences

A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow od data between the instructions associated with that name.

1. An *antidependence* between instruction *i* and *j* occurs when *j* writes a register or memory location that *i* reads. The original ordering must be preserved to ensure that *i* reads the correct value.
2. An *output dependence* occurs when instruction *i* and *j* write the same register or memory location. The ordering must be preserved to assure that the final value written corresponds to *j*

Name dependencies can be resolved by *renaming*.

Data Hazards

- RAW – read after write: *b* tries to read source before it is written by *a*, you now get the *old* value.
- WAW – write after write *b* writes an operand before it is written by *a*, you are writing in the wrong order
- WAR – write after read *b* tries to write to a location before it is read by *a*.

Control Dependences

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot move the then part of an if and move it before the if.
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot move the statement before the if and move it to the then part.

Dynamic Scheduling / Exceptions

Idea is simple; perform *out of order* execution, implying *out of order completion.* We can now have imprecise exception:

1. The pipeline may have *already completed* instructions that are *later* in program order than the instruction causing the exception.
2. The pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

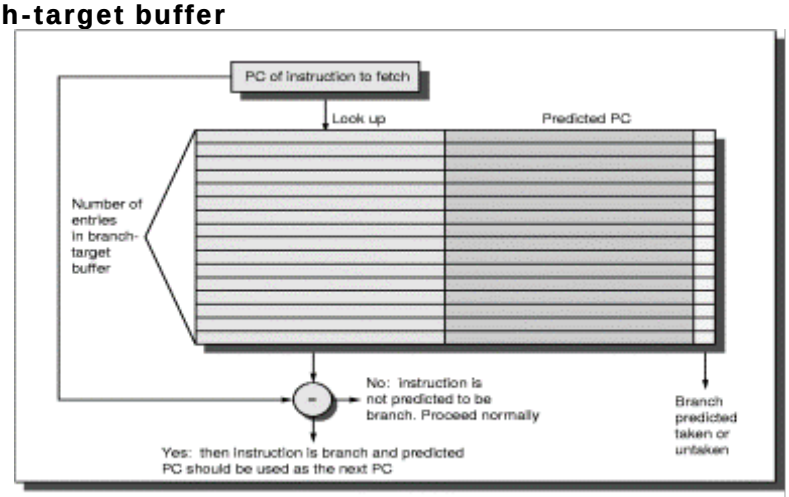
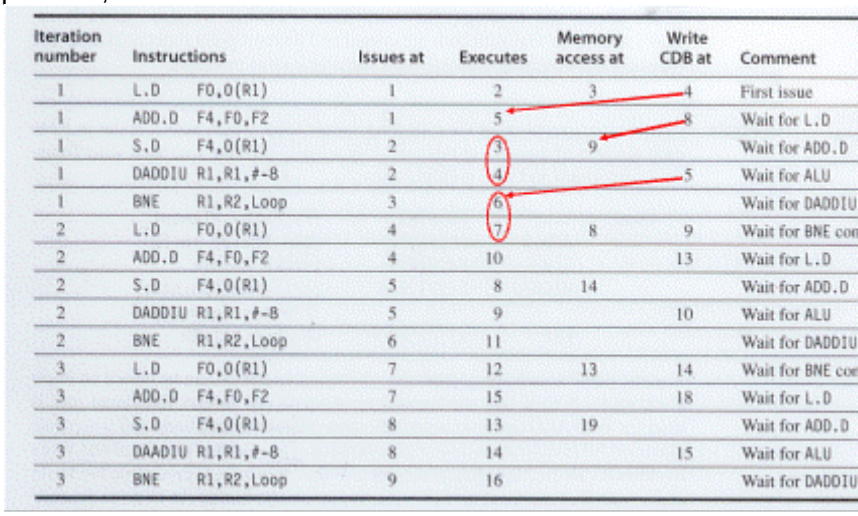
- Multiple Instruction Fetch/Issuing (Ch 4.3)
- **Branch-target buffer**
- 

FIGURE 4.22 A branch-target buffer.
- **Integrated instruction fetch unit**
  - Integrated branch prediction
    - Branch predictor is part of the fetch unit
  - Instruction prefetch
  - Instruction memory access and buffering
    - Fetching multiple instructions may require access of multiple cache lines.
  - Store target instructions instead of addresses
    - Saves on fetch time.
    - Permits branch folding - zero-cycle branches
      - Substitute destination instruction for branch in pipeline!
- **Return address predictors**
  - Predicting register/indirect branches
  - Generated from
    - abstract function calls
    - switch statements
    - procedure returns
  - BTB can be used to predict return addresses
    - What if the procedure is called from many places
      - CPU-internal return-address stack

Multiple Instruction Fetch/Issuing

- Issue multiple instructions in a single clock cycle.
  - Can get CPI < 1, or IPC > 1
- Two basic "flavors" of multiple-issue:
  - **Superscalar: [issue – dynamic]**
    - Maintain ordinary serial instruction stream format.
    - Instructions per clock (IPC) varies widely.
    - Instruction Issue can be dynamic or static (in-order).
    - **Multiple-issue Tomasulo:**
      - Issue 1 integer + 1 FP instruction to RS each cycle
      - Problem issuing multiple inst. simultaneously
        - If instructions dependent, hazard detection is complex.
      - Two solutions to this problem:
        - Enter inst. into tables in only 1/2 a clock
        - Build hardware to issue two instructions in parallel;
    - 

Iteration number	Instructions		Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D	F0,0(R1)	1	2	3	4	First issue
1	ADD.D	F4,F0,F2	1	5		8	Wait for L.D
1	S.D	F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU	R1,R1,#-8	2	4		5	Wait for ALU
1	BNE	R1,R2,Loop	3	6			Wait for DADDIU
2	L.D	F0,0(R1)	4	7	8	9	Wait for BNE con
2	ADD.D	F4,F0,F2	4	10		13	Wait for L.D
2	S.D	F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU	R1,R1,#-8	5	9		10	Wait for ALU
2	BNE	R1,R2,Loop	6	11			Wait for DADDIU
3	L.D	F0,0(R1)	7	12	13	14	Wait for BNE con
3	ADD.D	F4,F0,F2	7	15		18	Wait for L.D
3	S.D	F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU	R1,R1,#-8	8	14		15	Wait for ALU
3	BNE	R1,R2,Loop	9	16			Wait for DADDIU

  - **VLIW (Very Long Instruction Word) [issue – static]**
    - New format: Parallel instructions grouped into blocks.
    - Instructions per block are fixed (by block size).
    - Mostly statically scheduled by compiler.

- Speculative Execution (Ch 3.7)

Hardware Based Speculation

- Dynamic branch prediction chooses which instructions will be pre-executed.
- Speculation executes instructions conditionally early (before branch conditions are resolved).
- Dynamic scheduling handles scheduling of different dynamic sequences of basic blocks encountered.
- **Implementation:**
  - Separate the execution of speculative instruction from the committing of results permanently to registers/memory.
  - New structure called the reorder buffer (ROB) holds results of instructions that have executed speculatively (or non-speculatively) but cannot yet be committed (commit in order).

        - The reorder buffer represents non-programmer-visible temporary storage, like the reservation stations in Tomasulo's algorithm.

- Shared-Memory multiprocessor systems with coherent caches (Ch 6.3 – 6.6)

Types of Parallel Processing:

- SISD (Single Instruction Single Data)
    - Uniprocessors
- MISD (Multiple Instruction Single Data)
    - Multiple processors on a single data stream
        - No commercial prototypes. Can be thought of as successive refinement of a given set of data by multiple processors (units).
- SIMD (Single Instruction Multiple Data)
    - Simple programming model, low overhead, and flexibility
    - All custom integrated circuits
- MIMD (Multiple Instruction Multiple Data)
    - Flexible
    - Use off-the-shelf microprocessors

MIMD

- Two types
    - Centralized shared-memory multiprocessors
    - Distributed-memory multiprocessors
- Exploits thread-level-parallelism
    - The program should have at least n threads or processes for a MIMD machine with n processors
- Threads can be of different types
    - Independent programs
    - Parallel iterations of a loop (extracted by compiler)

Centralized Shared-Memory Multiprocessors

- Small number of processors share a centralized memory
    - Use multiple buses or switches
    - Multiple memory banks
- Main memory has a symmetric relationship to all processors and uniform access time from any processor
    - SMP: symmetric shared-memory multiprocessors
    - UMA: uniform memory access architectures
- Increase in processor performance and memory bandwidth requirements make centralized memory paradigm less attractive.

Distributed Memory Multiprocessors

- Distributing memory has two benefits
    - Cost-effective way to scale memory bandwidth
    - Reduces local memory access time.
- Communicating data between processors is complex and has higher latency.
- Two approaches for data communication
    - Shared address space (not centralized memory)
        - Same physical addr. refers to same memory location.

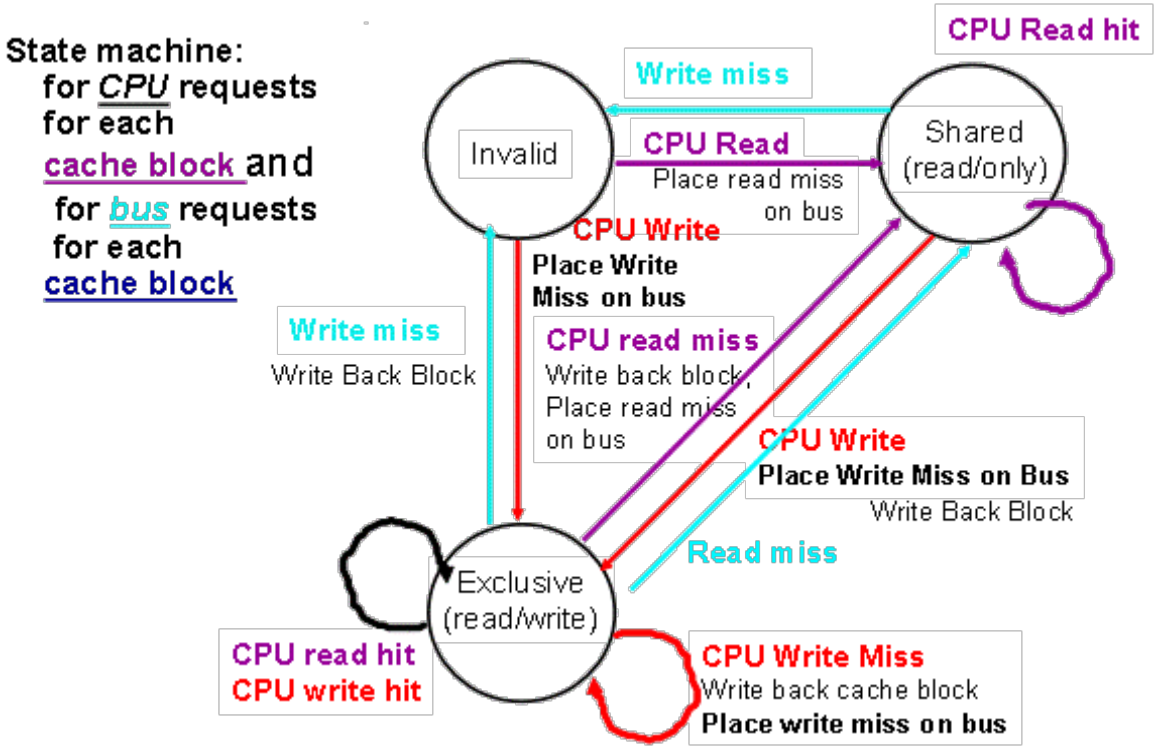
        - DSM: Distributed Shared-Memory Architectures
        - NUMA: Non-uniform memory access since the access time depends on the location of the data.
    - Logically disjoint address space - Multicomputers

Coherency:

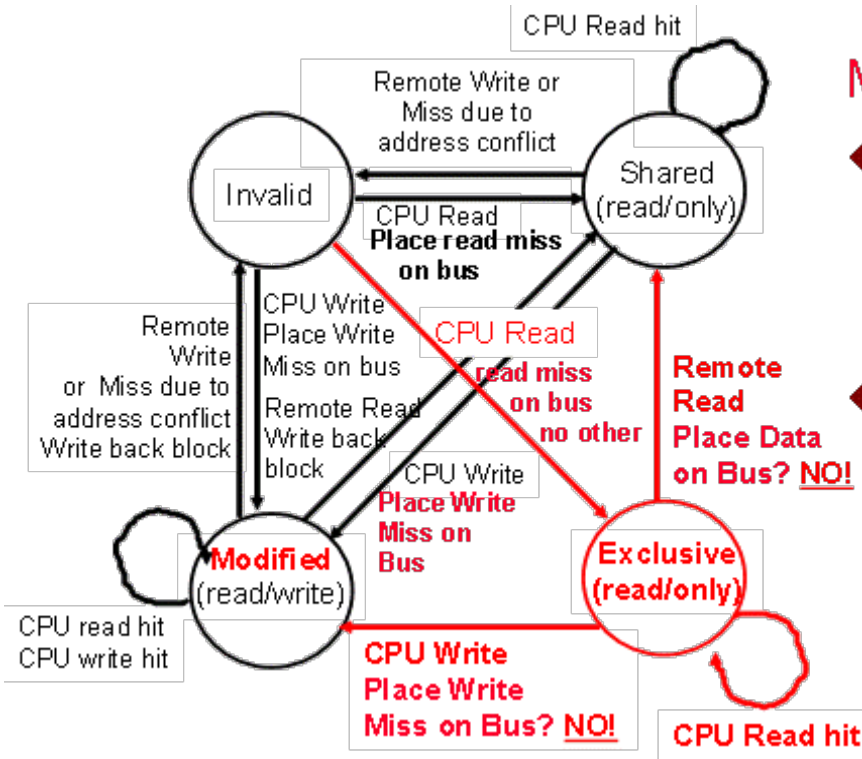
- Informally:
    - "Any read must return the most recent write"
    - Too strict and too difficult to implement
- Better:
    - "Any write must eventually be seen by a read"
    - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
    - "If P1 writes x and P2 reads it, P1's write will be seen by P2 if the read and write are sufficiently far apart" – far is defined by mem. consistency model
    - Writes to a single location are serialized:
        - Latest write will be seen

Coherency Solutions:

- Snooping Solution (Snoopy Bus)
    - Send all requests for data to all processors
    - Processors snoop to see if they have a copy and respond.
    - Requires broadcast, since caching info. is at processors
    - Works well with bus (natural broadcast medium)
    - Dominates for small scale machines (most of the market)
    - Snooping Protocols:
        - Write Invalidate Protocol
            - Multiple readers, single writer
            - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
            - Read Miss:
                - Write-through: memory is always up-to-date
                - Write-back: snoop in caches to find most recent copy
        - Write Broadcast Protocol (typ. write through)
            - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
            - Read miss: memory is always up-to-date
    - Basic Snooping Protocol:

- 

- MESI Protocol:



MESI Protocol:

- Clean exclusive state (no miss for private data on write)
- Exclusive state when read miss and no others

- Directory-Based Schemes
  - Keep track of what is being shared in 1 centralized place
  - Distributed memory => distributed directory for scalability
  - Send point-to-point requests to processors via network
  - Scales better than Snooping

- Actually existed BEFORE Snooping-based schemes