

Summary – Advanced Computer Architecture V 1.6

Index

Introduction.....	5
Computer Architecture.....	5
ISA.....	5
Computer Architecture concepts.....	5
Performance increasing.....	5
Self-aware computing.....	6
Metrics.....	6
Performance.....	6
Performance evaluation.....	6
Frequent case (Amdahl's Law).....	7
Corollary.....	7
Breaking down performances.....	7
CPU Time.....	7
MIPS and MFLOPS.....	8
MIPS.....	8
MFLOPS.....	8
Benchmarks.....	8
Average performances.....	8
Caches and Memory Hierarchy.....	8
Introduction to memory hierarchy.....	8
Memory Hierarchy.....	9
Principle of locality.....	9
Introduction to caches.....	9
4 Questions on caches.....	10
Block placement (where to put block in cache).....	10
Block identification (how block is found in cache).....	10
Mapping types.....	11
Direct mapped.....	11
N-Way associative.....	11
Fully associative.....	11
Block.....	11
Block replacement (which block to replace on a MISS).....	11
Write strategy (what happens during a write).....	11
Performance evaluation.....	12
Caches.....	12
Cache performances.....	12
Reduce miss rate.....	13
Reduce miss penalty.....	16
Reduce hit time.....	17
Memory technologies.....	18
Virtual memory.....	19

Virtual machine.....	19
Pipelining.....	19
Processors and ISA.....	19
ISA.....	19
Datapath vs Controller.....	19
MIPS architecture (Microprocessor without Interlocked Pipeline Stages).....	20
MIPS instructions.....	20
MIPS instructions execution.....	21
MIPS single/multiple cycles implementations.....	22
MIPS single-cycle implementation.....	22
MIPS multi-cycle implementation.....	23
Review of pipelining and MIPS.....	23
Pipeline performance.....	23
Optimized pipeline.....	23
Speed Up equation for pipelining.....	24
Issues.....	24
Hazards.....	24
Hazards classes.....	24
Structural.....	25
Control.....	25
Branch prediction techniques.....	26
Static Branch Prediction.....	26
Dynamic Branch Prediction.....	28
Data.....	30
Exception Handling.....	31
Interrupts.....	31
Exceptions classes.....	32
Asynchronous Interrupts.....	32
Interrupt handler.....	32
Synchronous Interrupts.....	33
5-stage pipeline exception handling.....	33
ILP – Instruction-level parallelism.....	33
Pipeline performance.....	33
ILP vs Parallel Processing.....	34
Type of Data Hazards.....	34
Issues in Complex Pipeline Control.....	34
Complex In-Order Pipeline.....	34
Increase Performance.....	35
Dependencies.....	35
Program properties for correctness.....	36
Dynamic and static scheduling.....	36
Static Scheduling (software).....	36
Loop unrolling.....	37
Trace Scheduling.....	38
VLIW (Very Long Instructions Word).....	40
VLIW instructions.....	40
VLIW machine.....	40
VLIW compiler.....	40
Pros and Cons.....	41
Loop execution → sw pipelining.....	41

VLIW instruction encoding.....	41
Problems.....	41
Problems.....	43
Dynamic Scheduling (hardware).....	43
Superscalar execution.....	43
Dynamic Scheduler.....	44
Basic data structure for instruction issuing.....	44
Scoreboard.....	44
Pro and cons.....	44
Summary.....	45
Scoreboard basic scheme.....	45
Exception handling.....	45
4 stages of Scoreboard Control.....	45
Scoreboard Structure.....	46
Scoreboard pipeline control.....	47
How To.....	47
Tomasulo Algorithm.....	48
Characteristics.....	48
Architecture.....	48
Components.....	48
Stages.....	49
Issue.....	49
Execution.....	49
WB.....	50
Details.....	50
Drawbacks.....	50
How To.....	50
Hardware Based Speculation.....	51
Reorder Buffer.....	51
Handling Exceptions.....	52
Register renaming.....	52
How to.....	52
Register Renaming.....	53
Scoreboard + Explicit Renaming.....	55
ILP Limits.....	55
Ideal machine.....	55
Limits on window size.....	56
3 models of memory alias.....	56
Multithreading and multicore processors.....	56
Parallel programming.....	57
Multi-threading.....	57
Multicore.....	58
Multicore and SMT.....	58
Parallel Architecture: SIMD and vector architectures.....	59
Parallel Architectures.....	59
Parallelism in applications.....	59
Hardware parallelism.....	59
Multiprocessor classification.....	59
SIMD.....	59
Vector Architectures.....	60

VMIPS architecture.....	60
MIMD.....	62
Shared Memory Architectures.....	63
Caching.....	64
Message Passing Architectures.....	68
Concurrency Management.....	68
Memory Consistency Problem.....	68
Synchronization.....	68
Release Consistency.....	68

This summary hasn't been revised by any professor

If you find any errors please contact me :)

By Flavio Primo

Introduction

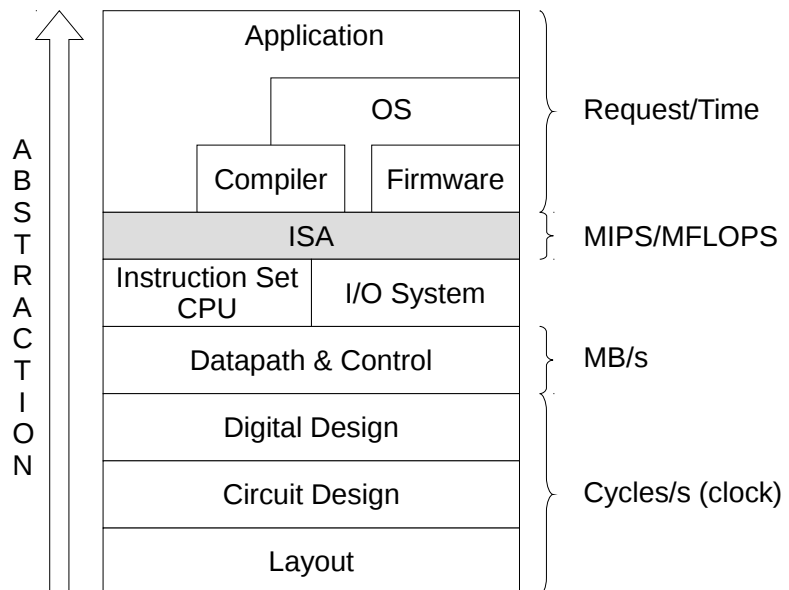
Computer Architecture

Computer Architecture:

design of abstraction layers that allow implementation of information processing efficiently using available manufacturing technologies.

It's an integrated approach:

- Coordination of abstraction levels
- Changes at every level
- Design, measurement and evaluation



ISA

Fundamental computer architecture component.

Instruction set is critical:

System attributes seen by a programmer

- organization of programmable storage
- data types/structures
- instruction set
- instruction formats
- modes for addressing and accessing data items/instructions
- exceptional conditions

Software
----- ISA
Hardware

Computer Architecture concepts

Performance increasing

Parallel architectures: collection of processing elements that cooperates (need communication architecture) and communicate to solve large problems fast → working in parallel instead of making faster processors.

Increase performance by:

- **internal parallelism:** pipelining, ILP, superscalar processors

- **process parallelism:** thread-level, multiprocessor/multicore architectures
- **cache memories:** addressing schemes

Self-aware computing

System that can observe their run-time behavior, learn and take actions to meet desired goals.

Characteristics:

- programmer specify goals → OS figures how to meet them
- System detects and handles errors as needed
- Translate resources into performances
- Increase sw adaptability and optimize for different platforms
- System is dynamic and adaptable to changing requests and operating conditions

Metrics

Driving measures for computer architecture: performance, power, costs.

Performance

- Performance of “x”: $Performance(x) = \frac{1}{t_{execution}(x)}$
- Speedup: “x” is n% time faster than “y”:
 $Speedup(x, y) = \frac{Performance(x)}{Performance(y)} = \frac{t_{execution}(y)}{t_{execution}(x)} = 1 + \frac{n}{100}$

Performance components:

Depends on:	IC (Instruction Count)	CPI (Clock Per Instruction)	Clock rate
Program	X		
Compiler	X	(X)	
Instruction set	X	X	
Organization	X	X	X
Technology			X

Performance evaluation

User: $t_{execution} = t_{end} - t_{start} \rightarrow \text{minimize}$ latency due to task completion: disk access, I/O, OS, ...	Center manager: $completion\ rate = \frac{\# \text{ jobs}}{t} \rightarrow \text{maximize}$ $throughput = \frac{1}{t_{avg\ response}} \text{ (if no overlap)}$
---	---

Performance factors:

- algorithm complexity and data set
 - operations
- } SW + HW

- compiler
- ISA
- OS
- clock rate system memory

Frequent case (Amdahl's Law)

Improve the speed on the most common case.

Speedup formula where y=old and x=new.

$$Speedup_{overall} = \frac{t_{execution}^{(old)}}{t_{execution}^{(new)}} = \frac{1}{1 - Fraction_{enhanced} + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$\text{where: } t_{execution}^{(new)} = t_{execution}^{(old)} \cdot (1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}$$

Corollary

If enhancement is only usable for a fraction of a task, we can't speed up the task by more than:

$$Speedup_{max} = \frac{1}{1 - Fraction_{enhanced}}$$

Considering costs:

$$Cost = Cost_{fraction_{unaffected}} + Cost_{fraction_{enhanced}} \cdot Cost_{enhanced}$$

if $Cost \gg Speedup_{overall}$ it's not worth it

Breaking down performances

CPU Time

Goal: minimize t_{CPU} time spent running program without considering I/O

$$t_{CPU} = \frac{s}{Program} = IC \cdot CPI \cdot t_{cycle} = \frac{Instructions}{Program} \cdot \frac{Clock\ cycles}{Instructions} \cdot \frac{s}{Clock\ cycles}$$

where:

$IC(Instruction\ count) = \frac{Instructions}{Program}$	Instructions actually executed (not static code). Determined by: algorithm, compiler, ISA
$CPI(Cycles\ per\ Instruction) = \frac{Clock\ Cycles}{Instructions} = \sum_{i=1}^n CPI_i \cdot Instruction\ Frequency_i$	Determined by: ISA and CPU organization (pipeline reduces CPI) $t_{cpu\%} = \frac{CPI_i}{CPI}$
$t_{cycle} = \frac{Seconds}{Clock\ cycles} \quad f_{clock} = \frac{1}{t_{cycle}}$	Determined by: technology, organization and circuit design

MIPS and MFLOPS

MIPS

Millions of instructions per second.

Higher MIPS → faster machines.

$$MIPS = \frac{Instructions}{t_{execution} \cdot 10^6} = \frac{f_{clock}}{CPI \cdot 10^6}$$

Important: only compare machines with same ISA, because IC changes performance ranking.

MFLOPS

Million floating point operations per seconds.

Higher MFLOPS → faster machines.

$$MFLOPS = \frac{FP \text{ operations in a program}}{t_{execution} \cdot 10^6}$$

Important: FP instructions may be missing in some ISA, compilers may optimize FP operations.

Benchmarks

Standard performance test programs. Different types are available:

- **Real programs:** like: compilers, apps, → accurate (not biased)
- **Kernel:** Code that represents group of similar programs → accurate (for selected features)
- **Synthetic:** matches the avg frequency of operations of a large set of sw → not reliable

Problems of benchmarks programs:

- I/O bound is a problem
- age poorly (vendors optimize architectures to fake benchmarks results)

Average performances

Averaging methods:

Total execution time comparison	$Performance = \frac{t_{executiontime}(B)}{t_{executiontime}(A)}$	
Arithmetic mean (times)	$Performance = \frac{1}{n} \sum_{i=1}^n t(i)$	if sw runs equally often
Weighted average (rates)	$Performance = \left\{ \sum_{i=1}^n t(i) \cdot weight(i) \right\} \div \frac{1}{n}$	if sw don't run equally often

Best to use unnormalized numbers.

Caches and Memory Hierarchy

Introduction to memory hierarchy

Memory becomes more and more important thanks to multi-core processors (which widen the bandwidth needs from data in memory):

Goal:

Illusion of **fast, large, cheap memory**: let programs address memory space that scales to disk size at a speed usually fast as register accessing.



Solution:

Memory hierarchy with different techs, costs, sizes and access mechanisms. **Faster** cache memory between CPU and DRAM.

Memory Hierarchy

Common memory hierarchies:

1. Register
 2. L1 Instr: contains program code (instructions, ...)
 3. L1 Data: contains program data (variables, ...)
 4. L2
 5. DRAM
 6. Disk
- $\left. \begin{array}{l} 2. \text{ L1 Instr: contains program code (instructions, ...)} \\ 3. \text{ L1 Data: contains program data (variables, ...)} \end{array} \right\} \text{Cache}$
 $\left. \begin{array}{l} 4. \text{ L2} \\ 5. \text{ DRAM} \\ 6. \text{ Disk} \end{array} \right\} \text{CPU}$

Principle of locality

Program access a small portion of address space at any instant of time. 2 “localities”:

- **Temporal**: if loc referenced → likely to be referenced again in the future (loops, reuse, ...)
 - cache stores contents of recently accessed loc
- **Locality**: if loc referenced → likely that locations near it will be referenced again (array, ...)
 - cache fetches blocks of data around recently accessed loc

Introduction to caches

Cache algorithm: read

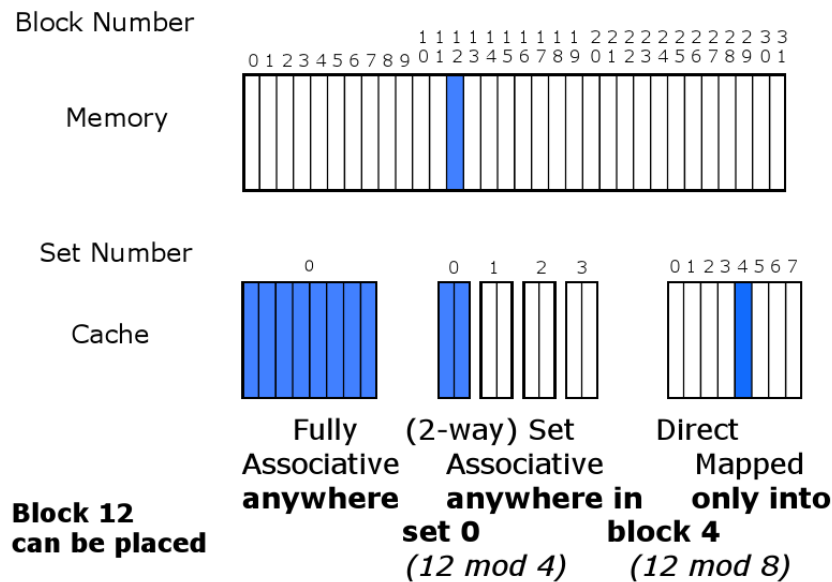
1. Look at cpu address
2. search cache tags to find match:
 - HIT: In cache → return copy of data from cache
 - MISS: Not in cache
 1. Read block of data from main memory
 2. wait
 3. return data to cpu & update cache

performance formulas:

- $f_{HIT} = \frac{1}{\text{found accesses in cache}}$
- $f_{MISS} = 1 - f_{HIT}$
- $t_{HIT} = t_{\text{cache access}} + t_{\text{determine HIT / MISS}}$
- $t_{MISS} = t_{\text{replace block in cache}} + t_{\text{deliver block to processor}}$

4 Questions on caches

Block placement (where to put block in cache)



Cache MISS sources:

- **Compulsory:** first access to a block never been in cache (cold start)
- **Capacity:** cache cannot contain all blocks accessed by the program
solution: increase cache size
- **Conflict (collision):** multiple memory locations mapped to same cache location
solution: increase cache size OR increase associativity
- **Coherence (invalidation):** other processes updates memory

Block identification (how block is found in cache)

1. "index" selects set (line)
2. "tag" gets block (minimum quantum of caching)

Increasing associativity → shrinks index & expands tag (fully associative cache don't have indexes).

Memory address:

Block address		Block Offset
Tag	Index	

Mapping types

<p>Direct mapped</p> <p>No collisions are possible, each index has its specific cache block (calculated with mod).</p>	<p>N-Way associative</p> <p>For the same index I can have n matches (n-way associative memory).</p>	<p>Fully associative</p> <p>No index is used, each memory address has it's own block.</p>

Block

Block is unit of transfer between the cache and memory

Large block size advantages and disadvantages:

Advantages:

- less block overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide buses

Disadvantages:

- more conflicts (less blocks)
- waste bandwidth

Block replacement (which block to replace on a MISS)

- **Direct Mapped:** easy
 - **Set Associative** or **Fully Associative:** various policies available
 - *LRU*: cache must be updated on every access, feasible only for small sets (2-way)
 - *pseudo-LRU*: binary tree often used (4-8 way)
 - *FIFO, LIFO*: for highly associative caches
 - *Random*: ok for big cache sizes
- } need special hw

larger the cache → the higher the associativity → the less difference between random and LRU.

Write strategy (what happens during a write)

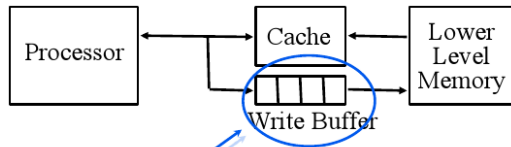
HIT

- **write through:** w cache+memory
high traffic but better coherence
debug: easy (can only see main memory which is synced in this case)
repeated write: make it to lower level

MISS

- **no write allocate:** w main memory only
- **write allocate:** fetch into cache (and update with write trough or write back)

write buffer:



Holds data awaiting write-through to lower level memory

cpu doesn't stall and supports write bursts, but RAW is a problem

- **write back:** w cache only (memory only when entry is evicted)

dirty bit per block can reduce traffic

debug: hard

read misses: produce writes

Common combinations:

- write through + no write allocate
- write back + write allocate

Performance evaluation

Caches

Cache performances

	f_{MISS}	$t_{MISS\ penalty}$	t_{HIT}	3C affected
↑ block size	↓	↑		↓ Compulsory misses
↑ cache size	↓		↑	↓ Capacity misses
↑ associativity	↓	↑	(↑)	↓ Conflict misses
multilevel caches		↓		
“r” priority over “w”			↓	

Compromise on:

- | | | | |
|---|---|----------------------|---|
| <ul style="list-style-type: none"> • cache size • block size • associativity | } | impact on each other | <ul style="list-style-type: none"> • w-through vs w-back • replacement policy |
|---|---|----------------------|---|

Cache performance equations:

- $t_{CPU} = (\text{CPU clock cycles} + \text{memory stall cycles}) \times t_{\text{clock cycle}}$ CPU stalls on cache MISS and clock cycles include time to handle cache HIT

- $\text{memory stall cycles} = \text{MISS} \cdot t_{\text{MISS penalty}} = IC \cdot \frac{\text{MISS}}{\text{instruction}} \cdot t_{\text{MISS penalty}} =$
 $IC \cdot r \text{ per instruction} \cdot f_{\text{MISS r}} \cdot t_{\text{MISS r penalty}} + IC \cdot w \text{ per instruction} \cdot f_{\text{MISS w}} \cdot t_{\text{MISS w penalty}}$
- $\frac{\text{MISS}}{\text{instruction}} = f_{\text{MISS}} \cdot \frac{\text{memory accesses}}{\text{instruction}}$
- $\text{AMAT}_{\text{average memory access time}} = t_{\text{HIT}} + f_{\text{MISS}} \cdot t_{\text{MISS penalty}} =$
 $(t_{\text{HIT inst}} + f_{\text{MISS inst}} \cdot t_{\text{MISS Penalty inst}}) + (t_{\text{HIT data}} + f_{\text{MISS data}} \cdot t_{\text{MISS Penalty data}})$
- $t_{\text{CPU}} = IC \cdot \left(\begin{array}{l} \left(\frac{CPI_{\text{execution}}}{\text{instruction}} + \frac{\text{memory access}}{\text{instruction}} \cdot f_{\text{MISS}} \cdot t_{\text{MISS penalty}} \right) \\ \left(\frac{CPI_{\text{execution}}}{\text{instruction}} + \frac{\text{memory MISS}}{\text{instruction}} \cdot t_{\text{MISS penalty}} \right) \\ \left(\text{ALUOps} \cdot \frac{CPI_{\text{ALUOps}}}{\text{instruction}} + \frac{\text{memory accesses}}{\text{instruction}} \cdot \text{AMAT} \right) \end{array} \right) \cdot t_{\text{clock cycle}}$

Reduce miss rate

Causes of cache misses: Compulsory, Capacity, Conflict

Rules of thumb:

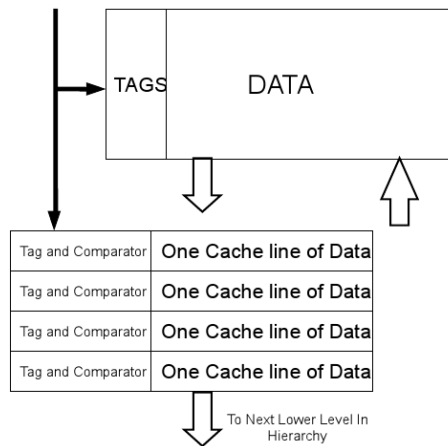
- 8-way set associative: as effective as full associative
- 2:1 cache: direct-mapped of size N == 2-way set associative of size N/2

Methods:

- \uparrow *cache size* but long hit time + energy consumption + cost
- \uparrow *associativity* but --
- \uparrow *block size* but increase miss penalty (reduced number of blocks in cache)
- *Multi-banked cache*: cache organized as independent banks for simultaneous access sequential interleaving between blocks.

Block address	Bank 0	Block address	Bank 1	Block address	Bank 2	Block address	Bank 3
0		1		2		3	
4		5		6		7	
8		9		10		11	
12		13		14		15	

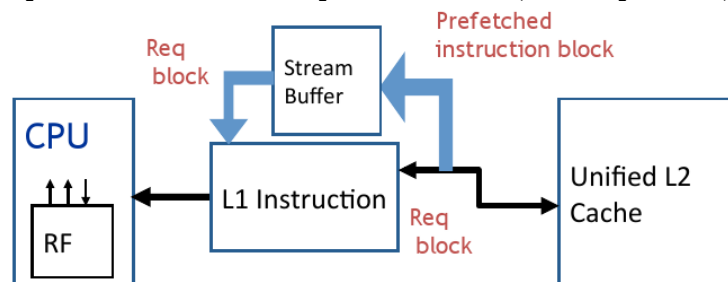
- *Victim cache*: small, fully-associative, LRU cache which assists a direct-mapped cache. Discarded cache ends up in the victim cache to exploit temporal locality. One extra clock cycle cost.



- *Pseudo associativity*: divide cache in 2: on miss check other half of cache, if so → pseudo-hit (slower hit). Combine fast hit time of direct-mapped and lower conflict misses of 2-way set associative cache. Not for caches tied to the processor (like the L2) because of variations in number of cycles in the pipeline.



- *HW prefetching instr/data*: instr/data can be prefetched directly in cache or in external stream buffers.
 - i-stream buffer (instr prefetching):
 - miss → fetch 2 blocks: i (in cache) and i+1 (in stream buffer)
 - hit in stream buffer (slow hit) → move i+1 (in cache) and fetch i+2 (in stream buffer)
 - d-stream buffer (data prefetching):
 - miss → fetch 2 blocks: b (in cache) and b+1 (in stream buffer)
 - hit on b → prefetch b+1 block (OBL - One Block Lookahead, can extend to N blocks)
 - observe hit pattern b, b+1, b+2 → prefetch b+3N (strided prefetch)



- *SW prefetching data*: compiler inserts prefetch instr to request data in advance. 2 types:
 - register prefetch: loads values in register
 - cache prefetch: loads values in cache
 but prefetch instructions are costly.
- *Compiler optimizations*:
 - instr: reorder procedures to reduce conflict misses and profiling to look at conflicts
 - data: improve spatial locality
 - **merge arrays** (single array of compound elements)

```

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];
/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];


```

- **loop interchange:** change nesting of loops

```

/* Before */
for (k = 0; k < 100; k = k+1)
    for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
    for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];

```



- **loop fusion:** combine 2 independent loops that have same looping and some vars overlap

```

/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        { a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j]; }

```

- **blocking:** improve temporal locality by accessing blocks of data repeatedly

<pre> /* Before */ for (<u>i</u> = 0; i < N; i = i+1) for (j = 0; j < N; j = j+1) { r = 0; for (k = 0; k < N; k = k+1) { r = r + y[<u>i</u>][k]*z[k][j]; } x[<u>i</u>][j] = r; } </pre>	<pre> /* After */ for (jj = 0; jj < N; jj = jj+B) for (kk = 0; kk < N; kk = kk+B) for (i = 0; i < N; i = i+1) for (j = jj; j < min(jj+B-1,N); j = j+1) { r = 0; for (k = kk; k < min(kk+B-1,N); k = k+1) { r = r + y[i][k]*z[k][j]; } x[i][j] = x[i][j] + r; } </pre>
--	--

Reduce miss penalty

- **Read Priority over Write on Miss:** serves read before writes have been completed by giving higher priority to read misses over writes.
 - *Write-through cache + write buffer*
 Problem: write buffer might hold the updated value of a location needed on a read miss (RAW). Solutions:
 - wait until write buffer is empty (but increases the MISS penalty)
 - check write buffer for the updated content
 - *Write-back:* instead of writing the dirty block to memory → copy dirty block to a buffer then read and then write memory
- **Sub-block placement:** add “valid bit” to units smaller than a full block (sub-block), only sub-block is read on a miss. Sub-block have smaller penalty than full blocks.
- **Early restart and critical word first on miss:** cpu normally needs only one word from a block, so 2 strategies for early loading of words:
 - *critical word first:* request missed word first, then while executing read rest of the block
 - *early restart:* fetch words in normal order and when read the requested word send it immediately to the processor.

Benefits of this approach depend on the side of the block.

- **Non-blocking caches:** “non-blocking cache” allows data to continue to supply cache hits during a miss (requires: out-of-order execution cpu, multi-bank memories).
 - “Hit under miss” reduces the effective miss penalty by working during miss (instead of ignoring cpu requests). Better mechanism if it can overlap multiple misses but complex cache controller that needs to manage multiple memory accesses.

Effective miss penalty is the effect of the non overlapped time that the cpu is stalled.

- **Multilevel caches:** second level of cache that support the misses of the primary cache.

2-level cache allows for:

- primary cache: minimizes hit time to yield shorter clock cycle or fewer pipeline stages
- secondary cache: minimizes miss rate to reduce the penalty of long memory access times. Not tied to cpu clock rate.

Use simpler write trough L1 → L2 and a smaller L1

$$AMAT_{\text{Average memory access time}} = t_{\text{HIT L1}} + f_{\text{MISS L1}} \cdot t_{\text{MISS penalty L1}}$$

$$t_{\text{MISS penalty L1}} = t_{\text{HIT L2}} + f_{\text{MISS L2}} \cdot t_{\text{MISS penalty L2}}$$

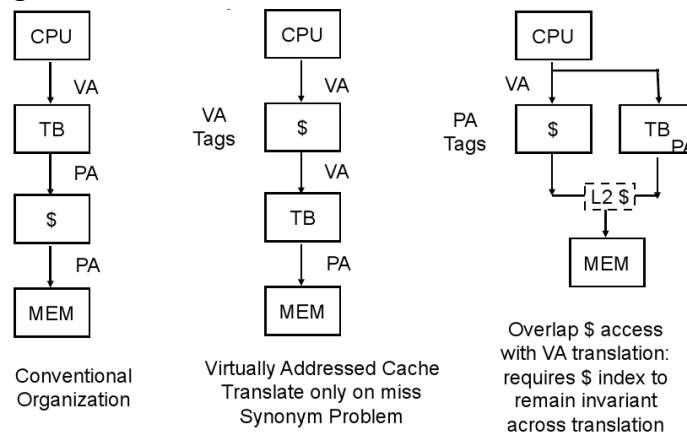
- local miss rate: number of misses in a cache divided by the total number of memory accesses to this cache ($f_{\text{local MISS L1}} = f_{\text{MISS L1}}$, $f_{\text{local MISS L2}} = f_{\text{MISS L2}}$)
- Global miss rate: number of misses in the cache divided by the total number of memory accesses generated by the cpu ($f_{\text{global MISS L1}} = f_{\text{MISS L1}}$, $f_{\text{global MISS L2}} = f_{\text{MISS L1}} \cdot f_{\text{MISS L2}}$)

L1 can either only include elements in L2 (inclusive ml cache) or include also other elements (exclusive ml cache)

Reduce hit time

- **Small & Simple caches:** critical time path in a cache hit is the 3-step process:
 1. address tag memory using index portion of the address
 2. compare read tag value to the address
 3. setting multiplexor to choose the correct data item if cache is set associative
 smaller cache lowers the hit time (less time to check for entries).
- **Fast hits by avoiding address translation:** when the virtual memory addresses are used also for the cache. Excellent because eliminates the address translation time, but several problems arise:
 - Protection, page-level protection is checked as part of virtual to physical address translation and must be enforced → solution: get protection info from TLB on a miss and copy it on a dedicated field to check it at every access to virtually addressed cache.
 - Every time process switch happens, need to flush the cache → solution: increase address tag with the PID (need flush only if PID recycled)
 - OS and user programs may use different virtual addresses generating aliases → solution: SW lower n bits have same address (page coloring) or HW unique physical address for each block
 - I/O which uses physical addresses

Best: “virtually indexed, physically tagged” in which caches are indexed with the offset which remains the same in virtual and physical address, allowing the indexing to begin (hit) before virtual address translation is completed. Problem: direct-mapped cache cannot be bigger than page size.



Cache indexing is time critical, since all the lines in a set can be read in parallel and the appropriate line selected based on a tag comparison.

- **Fast hit times via pipelined writes:** pipeline cache tag check and update cache data as separate stages → promotes high associativity, but high penalty on mispredicted branches and more cycles to load/use data
- **Fast writes on misses via small sub-blocks:** if most writes are 1 word then sub-block size of one word and write through then write sub-block and tag immediately:
 - tag match and valid bit already set → w ok and nothing lost by setting valid bit on again
 - tag match and valid bit not set → proper block, w makes turn valid bit on

- tag mismatch → miss that will modify data portion of the block (for this reason doesn't work for write back)

Memory technologies

	SRAM (Static Random Access Memory)	DRAM (Dynamic Random Access Memory)
<i>Usage</i>	cache	main memory
<i>Transistor/bit</i>	6	1
<i>Bit retention technique</i>	low power	periodically refreshed
<i>Addressing</i>	Direct mapped fully associative 2-way associative	address lines multiplexed: RAS (Row Access Strobe): upper half of address CAS (Column Access Strobe): lower half of address
<i>Optimizations</i>		<ul style="list-style-type: none"> • Multiple access on same row (read more sequential words on the same row) • Synchronous (add clock to DRAM interface) • Wider interfaces • DDR (double data rate): provide 2 accesses on same clock cycle • multiple banks on each DRAM device

Amdahl: Memory capacity should grow linearly with CPU speed (not happening)

GRAPHIC MEMORY

- higher clock rate
- wider interface (32bit)
- 2-5x bandwidth wrt to DRAM

SDRAM (Synchronous Dynamic Random Access Memory)

DRAM where the external operations are managed by external clock. Power optimization:

- Lower voltage
- Low power mode (ignore clock, continue to refresh)

FLASH MEMORY

- EEPROM
- overwrite: must erase blocks before
- non volatile
- limited number of w cycles
- cheaper and slower than SDRAM

Memory dependability:

- soft errors: dynamic errors → solve with ECC
- hard errors: permanent errors → solve with spare rows to replace defective rows
- chipkill: RAID like recovery technique (words in different chips)

Virtual memory

- Protection: processes confined in own memory space
- Architecture:
 - provides and manages user and supervisor mode
 - protects some aspects of CPU states
 - limits memory accessing
 - TLB to translate addresses

Virtual machine

Security through resources isolation.

Allows different ISA and OS to software.

VMM let each VM OS to maintain its own set of page tables:

- adds “real memory” level between physical and virtual memory
- maintains shadowpage table that maps virtual machine addresses to physical addresses

Pipelining

Processors and ISA

ISA

Defines: set of operations, instruction format, supported hw, data types, named storage, addressing mode, sequencing.

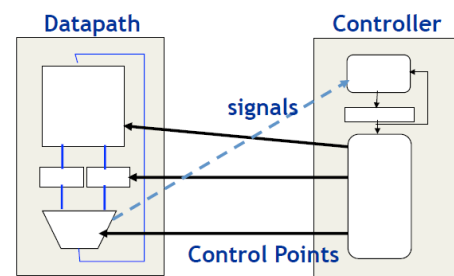
- RTL (Register Transfer Language): design abstraction that describe each instruction meaning
- assembles datapath with given technologies (FU, storage mapping from design to actual, ...)
- maps instruction to an RTL sequence
- implements controller

Datapath vs Controller

Datapath: storage, FU, interconnect resources to perform operations.

Contains: ALU, registers, ...

Controller: a state machine that controls the operation's



flow of execution on the datapath.

Contains:

- *PSW (Program Status Word)*: status register that contains a flag indicating the arithmetic operation status.
- *IR (Instruction Register)*: contains the instruction being performed.
- *PC (Program Counter)*: contains next instruction reference.

Datapath and controller are connected to the memory through the bus, but may be necessary more than one transfer because the bus size may be smaller than the number of the registers.

MIPS architecture (Microprocessor without Interlocked Pipeline Stages)

- **RISC ISA**: based on simple instructions in a reduced basic cycle to optimize the performance of CISC CPUs.
 - Format instruction: 32bit fixed (3 formats) → few instr format and all have same length
 - GPR (General Purpose Registers): x32 32bit (R0 contains 0, DP take pair)
 - Instruction: 3 address, address arithmetic is reg-reg
 - LOAD/STORE address: base+displacement (no indirection aka pointers)
 - Branch conditions: simple, delayed branch
- **LOAD/STORE**: ALU operands come from the CPU general purpose registers (and not from memory). Dedicated instruction for: load data from memory to registers, store data from registers to memory.
- **Pipeline architecture**: performance optimization technique based on overlapping of execution of multiple instructions derived from a sequential execution flow.

MIPS instructions

	6bit	5bit	5bit	5bit	5bit	6bit
Reg-Reg	Op	Rs1	Rs2	Rd		Opx
Reg-Imm	Op	Rs1	Rd	Imm		
Branch	Op	Rs1	Rs2/Opx	Imm (range where to jmp)		
Jump/Call	Op	Tgt				

- ALU instructions:
 - `add $s1, $s2, $s3 # $s1 ← $s2 + $s3`
 - `addi $s1, $s1, 4 # $s1 ← $s1 + 4`
- LOAD/STORE instructions:
 - `lw $s1, offset ($s2) # $s1 ← M[$s2+offset]`
 - `sw $s1, offset ($s2) # M[$s2+offset] ← $s1`
- Branch instructions to control the control flow of the program:

- Conditional branches: branch taken iff condition is satisfied.
 - `beq $s1, $s2, L1` # branch-on-equal: go to PC+L1 if ($\$s1 == \$s2$)
 - `bne $s1, $s2, L1` # branch-on-not-equal: go to PC+L1 if ($\$s1 != \$s2$)
- Unconditional jumps: the branch is always taken:
 - `j L1` # jump: go to L1!
 - `jr $s1` # jump-register: go to add. contained in \$s1 (function calls)

MIPS instructions execution

Every MIPS instruction can be implemented in at most 5 clock cycles.

1. IF – Instruction Fetch

Send content of the PC register to IM (Instruction Memory) and fetch current instruction from IM (1clk if it's already in cache → cache HIT).

Update PC to next sequential address adding +4 (each instruction 4 Bytes).

2. ID – Instruction Decode

Decode IR (where resides the current instr) and read RF (Register File) of 1-2 registers depending if current instruction is reg-imm or reg-reg.

Sign-extension of the offset field of the instruction in case it is needed (example: if imm is 16bit long, need to extend to 32bit).

If “Early Evaluation” compare regs, compute and update PC.

3. EX – Execution Cycle

ALU operates on the given operands depending on the instruction type:

- *reg-reg ALU*: ALU executes specified operation on the operands read from the RF
- *reg-imm ALU*: ALU executes specified operation on the first operand read from the RF and the sign-extended immediate operand
- *Memory reference*: ALU adds base register and offset to calculate the effective address
- *Conditional branches*: compare 2 registers read from the RF and compute the possible branch target address by adding the sign extended offset to the incremented PC

4. MEM – Memory Access (only for LOAD/STORE/BRANCH)

LOAD: require “r” access to DM (Data Memory) using the effective address.

STORE: require “w” access to DM using the effective address. Writes data from “Source Register” of RF.

BRANCH: update PC (only if not Early Evaluation)

5. WB – Write-Back Cycle (only for ALU/LOAD)

Only phase where the CPU changes the CPU state.

LOAD writes data in the “Destination Register” of RF.

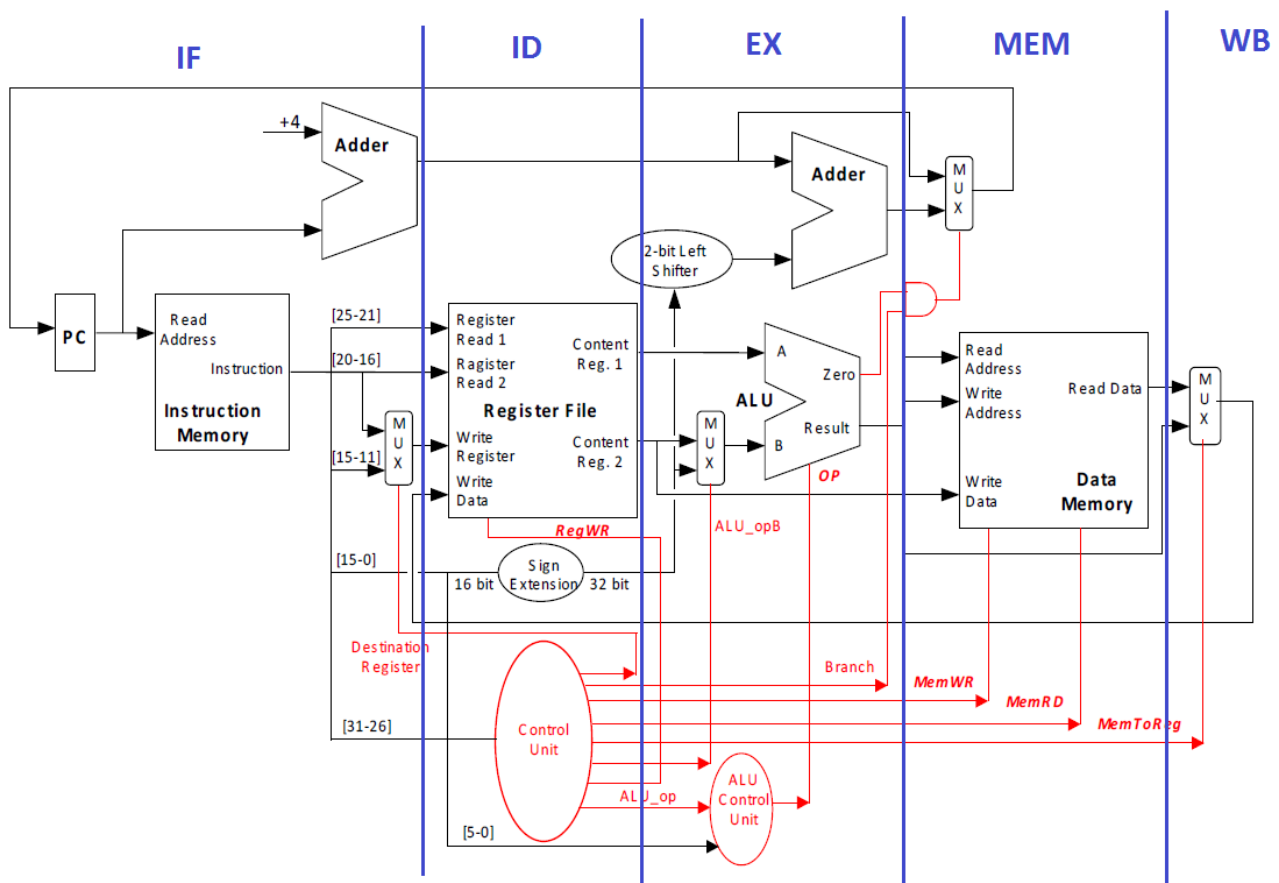
ALU instruction writes ALU results into “Destination Register” of RF.

MIPS instructions execution:

IF	ID	EX	MEM	WB
----	----	----	-----	----

<i>ALU</i> op \$x, \$y, \$z	Instr fetch + PC increm	Read source reg: \$y, \$z	ALU op \$y op \$z		WB dest reg \$x
<i>LOAD</i> lw \$x, offset(\$y)	Instr fetch + PC increm	Read Base reg: \$y	ALU op \$y+offset	Read mem M(\$y+offset)	WB dest reg \$x
<i>STORE</i> sw \$x, offset(\$y)	Instr fetch + PC increm	Read Base Reg: reg: \$y source: \$x	ALU op \$y+offset	Write mem M(\$y+offset)	
<i>COND BRANCH</i> beq \$x, \$y, offset	Instr fetch + PC increm	Read source reg: \$x, \$y	ALU op & (PC+4+offset)	Write PC	

MIPS datapath:



blue lines are interstage pipeline registers which store all data and control signal to free the stages

MIPS single/multiple cycles implementations

MIPS single-cycle implementation

Assume each instruction executed in a single clock cycle.

- Clock cycle length defined by the critical path given by LOAD instruction
- IM and DM are separated
- Multiplexer enables module sharing between different instructions

MIPS multi-cycle implementation

Assume each instruction executed in multiple clock cycles (one clock for each phase).

- Cycle is shorter (higher throughput wrt single-cycle implementation)
- 5 cycles for MIPS arch
- Internal registers to store the values to be used in the next clock cycles

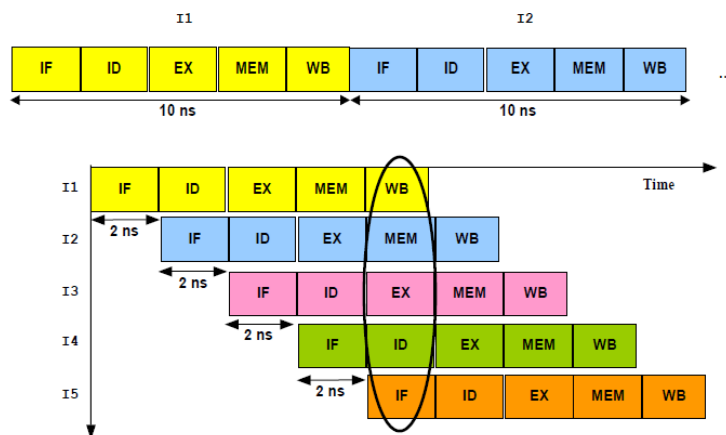
Review of pipelining and MIPS

Pipeline: performance optimization based on the overlap of the execution of multiple instructions deriving from a sequential execution flow.

Concept: instructions are divided in atomic operations (pipeline stages), the stages are connected one to the next to form a pipeline.

Pro:

- transparent to the programmer
- completion time of one instruction is the same as if there was non pipeline
- pipeline increase the throughput of the completed instructions



Characteristics:

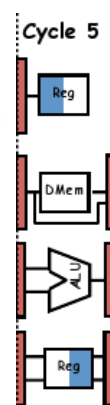
- time to advance in the pipeline: 1 clock
- pipeline stage are synchronized to the slowest operation (in terms of cycles) in the pipeline
- goal: balance length of pipeline stage (each stage has same duration). If perfectly balanced $speedup_{from\ pipeline} = \# \text{ pipeline stages}$
- interstage registers store intermediate results from each stage for the next stage

Pipeline performance

Optimized pipeline

RF used in 2 stages: Read access (ID) and write access (WB), possible solutions:

- “stall”
- “w” in 1° half and “r” in the 2° half of the clock cycle: w (rising edge) |__|__ r (falling edge)



Speed Up equation for pipelining

Speedup

CPI: Cycles per Instruction

$CPI_{\text{pipelined}} = \text{ideal CPI} (=1) + \text{avg stall cycles per instructions}$, $\text{ideal CPI} = 1$ for RISC pipeline

$$\text{Speedup}_{\text{pipelined}} = \frac{\text{ideal CPI} \cdot \# \text{ pipeline stages} \cdot t_{\text{cycle UNpipelined}}}{\text{ideal CPI} + \text{pipeline stall CPI} \cdot t_{\text{cycle pipelined}}}$$

if $\# \text{ pipeline stages} = \text{pipeline stall CPI}$ no advantage

Real Speedup (considers whole avg execution time)

$$\text{Real Speedup}_{\text{Pipeline}} = \frac{t_{\text{AVG EXEC UNpipelined}}}{t_{\text{AVG EXEC Pipelined}}} = \frac{\text{AVG CPI UNpipelined} \cdot t_{\text{clock UNpipelined}}}{\text{AVG CPI pipelined} \cdot t_{\text{clock pipelined}}}$$

Issues

Pipeline increases number of instr completed per unit of time (throughput), but it does not reduce the execution time (latency).

Pipeline introduce slight latency of each instruction: overhead (for pipeline control) + pipeline stages imbalance (longest stage is taken as clock time).

If $t_{\text{cycle Pipelined}} = t_{\text{cycle UNpipelined}}$ and stages balanced:

$$\text{Speedup}_{\text{Pipeline}} = \frac{\text{AVG CPI UNpipelined}}{\text{ideal CPI} + \text{Pipeline stall CPI}} = \frac{\# \text{ pipeline stages}}{1 + \text{Pipeline stall CPI}}$$

with:

- $\text{ideal CPI} = 1$ (RISC pipeline)
- $\text{AVG CPI UNpipelined} = \# \text{ pipeline stages}$ (all instr have same number of cycles that must equals number of pipeline stages)
- $\text{Pipeline stall CPI} = f_{\text{branch}} \cdot t_{\text{branch penalty}}$ for control hazard

if no pipeline stalls (ideal case) → improve performance by depth of pipeline

$$\text{MIPS} = \frac{f_{\text{clock}}}{\text{CPI} \cdot 10^6} \quad \text{with} \quad \text{CPI} = \frac{\overbrace{CI + \# \text{stall cycles} + 4}^{\# \text{clock cycles}}}{\underset{5 \text{ stage pipeline}}{CI}}, \quad CI = \# \text{ instructions}$$

Hazards

When a dependency between instructions exists and are close enough to overlap in the pipeline.

Hazards problems:

- prevent next instr execution in the pipeline from executing during its designated clock cycle
- reduce performance from the ideal speedup gained by pipelining

Hazards classes

- **Structural:** attempt to use same resource from different instructions simultaneously.
- **Data:** attempt to use result before it's ready.

- **Control:** attempt to make a decision on the next instruction to execute before the condition is evaluated.

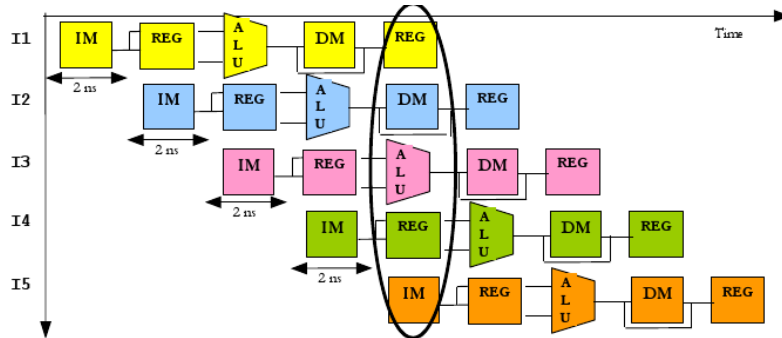
Structural

Attempt to use same resource from different instructions simultaneously

example: same memory for instructions and data.

MIPS doesn't have structural hazards:

- IM and DM separated
- RF used in the same clock cycle, with "r" and "w" performed by two different instructions



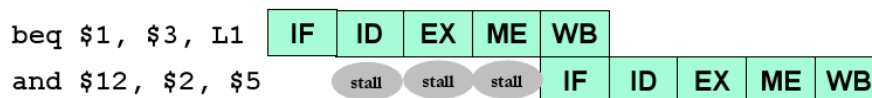
Control

Attempt to make a decision on the next instruction to fetch before the condition is evaluated.

example: conditional branch execution.

Conditional branch instruction: branch is **taken** (stored in the PC) iff condition is satisfied otherwise **untaken**.

- Branch outcome and Branch Target Addr are ready at the end of stage 3 (EX)
- Conditional Branches are solved when PC is updated at the end of stage 4 (MEM)

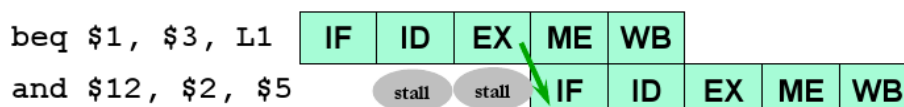


Problem: Control hazards reduce performance from the ideal speedup gained by the pipelining since they can make it necessary to **stall** the pipeline.

Need to fetch a new instruction at each clock cycle, but branch decision is taken during MEM stage → delay (called Control Hazard).

Solutions:

- **Forwarding:** since Branch outcome and Branch Target Addr are ready at stage 3 (EX) → forward the info (2 clock cycles instead of 3), while the outcome is known at ID stage.



Each branch costs 2 stalls or 2 instructions flush if branch is taken (2 clock cycles instead of 3).

- **PC early evaluation (better):** add hw resources (move addr adder back to ID stage) to:

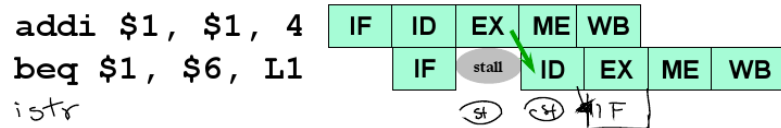
1. compare registers to derive branch outcome
2. compute branch target addr
3. update PC register

asap in the pipeline
in MIPS performed during stage 2 (ID)

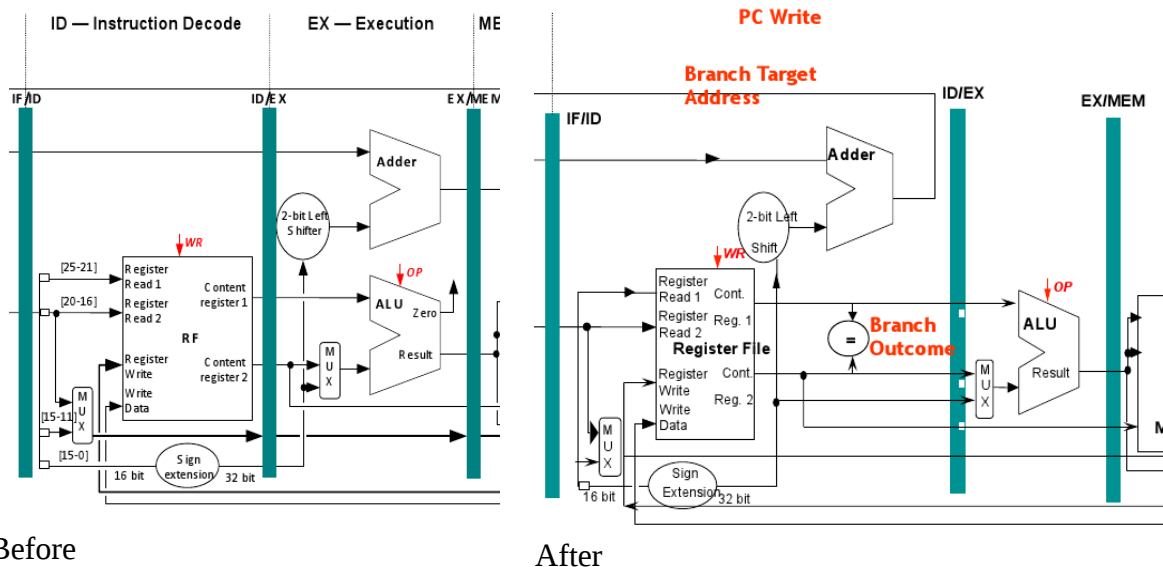
Each branch costs 1 stall or 1 instruction flush if branch taken (1 clock cycle instead of 3).



issue:



RAW can arise if need to perform evaluation on results of a previous instr.



Branch predictions more important for deep pipelines because the cost of incorrect predictions increases (branches solved 4 stages after ID).

Performance depends on:

- **Accuracy:** measured as percentage of incorrect predictions
- **Cost:** of incorrect prediction, measured as time lost to execute useless instructions
- **Branch frequency:** frequency of branch instructions in a program

Branch prediction techniques

- **Static:** actions for each branch are fixed at compile time, thus for the entire execution
- **Dynamic:** actions for a branch can be changed during the execution

Static Branch Prediction

Actions for each branch are fixed at compile time.

Use cases: sw with predictable branching at compile time, assist dynamic branch prediction.

Techniques:

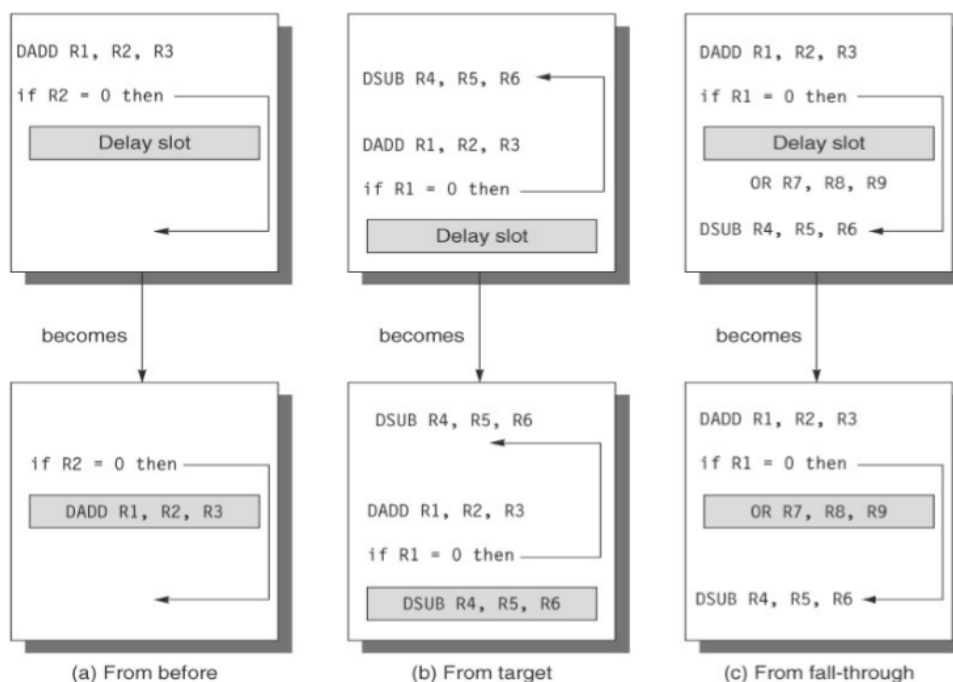
- **Branch Always Not Taken:** assume branch wont be taken.

- If condition not satisfied (ID) → performance saved
- If condition satisfied (ID) → branch is taken, need to flush next instruction already fetched (becomes a nop). 1 clock cycle penalty.
- **Branch Always Taken:** assume branch is taken. Ok if we know branching before results of current instruction (no in MIPS, so no advantages).
- **Backward Taken Forward Not Taken (BTFNT):** prediction based on branch direction:
 - if backward prediction (example: repeat a loop): assume as taken
 - if forward prediction (example: if-else): assume as not taken
- **Profile-Driven Prediction:** prediction based on previous runs. Can use compiler hints.
- **Delayed Branch:** compiler schedules an independent instruction in the branch delay slot which is executed whether branch is taken or not.

Behavior the same whether the branch is taken or not.

3 ways to schedule the branch delay slot:

- “*From Before*” (*best alternative*): instr in the delay slot is taken from before the branch (always executed)
- “*From Target*”: instr in the delay slot is taken from the taken branch (when branch is likely to be taken like in do-while loops). Usually copy instr because it can be reached from another path.
- “*From Fall-Through*”: instr in the delay slot is taken from the not taken branch (when branch is not likely to be taken like in if-else). Usually copy instr because it can be reached from another path.



Problems:

- The restrictions on the instructions that can be scheduled in the delay slot.
- In deep pipelines is more difficult to fill all the slots (need more instructions to fill the

delay slots).

- The ability of the compiler to statically predict the outcome of the branch.

Solution: introduce “canceling or nullifying branch”, instr includes the direction that the branch was predicted. If executed as predicted ok, otherwise NOP

MIPS arch: “branch-likely” instr that behaves as “cancel-if-not-taken” branch.

- Instr in branch is executed whether the branch is taken or not.
- Branch instr is canceled if the delay slot is not executed (turned into nop) whether the branch is untaken.

Dynamic Branch Prediction

Decision causing branch prediction can change during sw execution.

Based on 2 interacting mechanisms used by IF:

- | | |
|--|--|
| <ul style="list-style-type: none"> • <i>Branch Outcome Predictor (or BHT – Branch History Table)</i>: predicts branch direction (taken/not taken) • <i>Branch Target Predictor</i>: predicts branch target address in case of branch taken | <div style="font-size: 3em; vertical-align: middle; padding: 0 10px;">}</div> <div> <p>PC incremented</p> <p>else</p> <p>BTP gives the target address</p> </div> |
|--|--|

BTP (Branch Target Predictor)

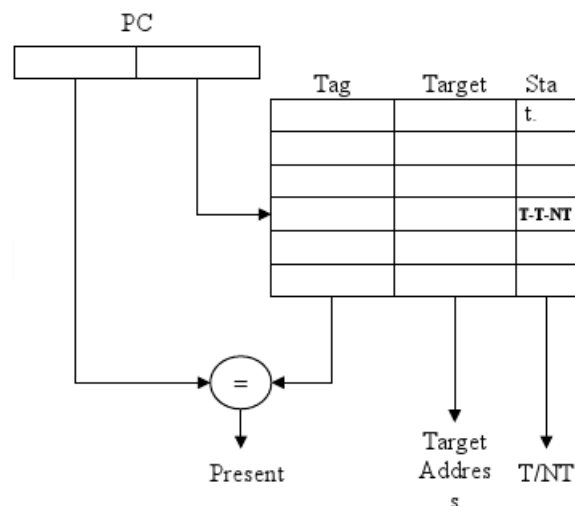
BTB (Branch Target Buffer): Cache that stores predicted branch target address for the next instruction after a branch.

If after getting the current instruction “branch”, takes its lowers bit of the PC and looks for it in the BTB. If found it then use the correspondent predicted branch address.

BTB entry: tag + predicted tgt addr + (prediction state bit)

- tag: indexed by lower portion of the address
- *predicted tgt addr*: points to branch taken (PC relative)
- (*prediction state bit*): 1bit for each entry that says if the branch was recently taken or not

Misprediction: if prediction incorrect or same index referenced by 2 different branches and the previous history refers to the other branch (→ increase #rows or hashing).



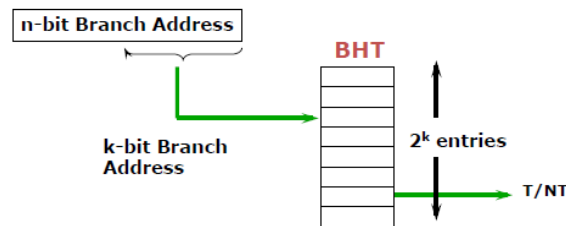
BHT (Branch History Table)

Branch can “come back” so we keep track of previous decisions.

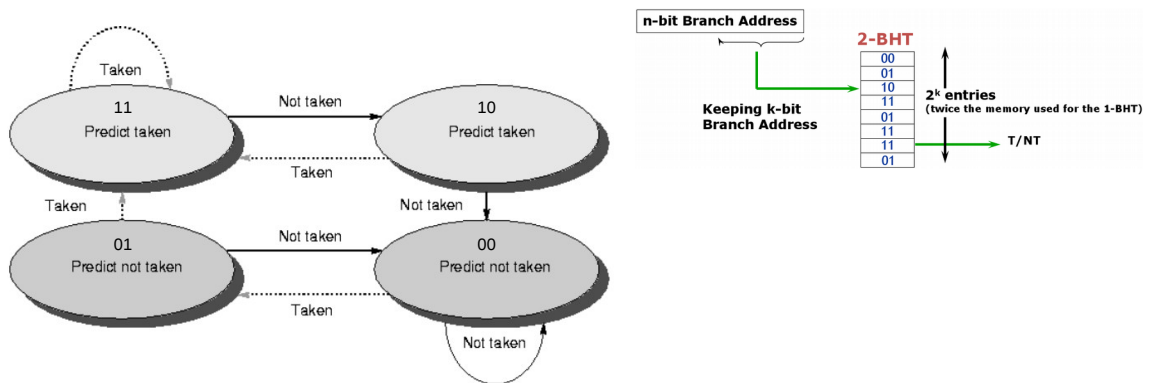
BHT indexed by lower portion of the address of the branch instruction.

Table starts initialized (always taken/not taken) and gets updated with a strategy:

- **1-BHT:** 1 misprediction → change predictions



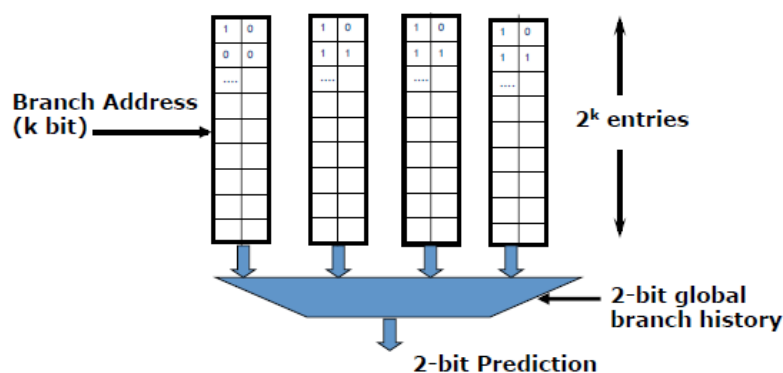
- Always taken: BHT initialized with all 1
 - loop 90% accuracy → last prediction wrong
 - nested loop 80% accuracy → except for the first (1 prediction), at each outer iteration we get 2 wrong predictions
- Always not taken: BHT initialized with all 0 (in a loop 80% accuracy → first and last predictions are wrong)
 - loop 80% accuracy → first and last predictions wrong
 - nested loop 80% accuracy → at each outer iteration we get 2 wrong predictions
- **2-BHT:** 2 misprediction → change predictions (entries encode the state of a 4 state FSM)



- **n-BHT:** n misprediction → change predictions (behaves as 2-BHT)
- **Correlating Branch Predictors**

Different from BHT, decision based on a different branches outcome. Decision based on selecting the correct 1-BHT.

(m,n) correlating predictor: where 2^m is the number of BHT to choose from and n is the n-BHT.



- *2-level Adaptive Branch Predictors*

Branch History Registers: stores 1° level history in 1/more k-bit shift registers, records outcomes of the last k branches. BHT indexes the PHT to select the correct one.

Pattern History Table: stores 2° level history in 1/more tables 2-bit counters

Data

Attempt to use a result before it is ready.

- **RAW (Read After Write)**

Instr J tries to read operand before previous instr I writes it → J reads wrong value

I: add **r1**, r2, r3

J: sub r4, **r1**, r3

Solutions:

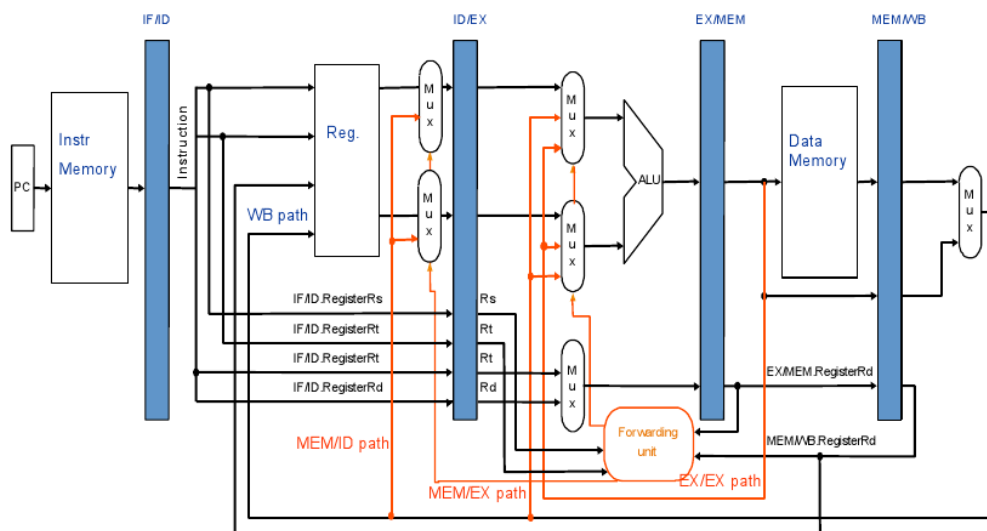
- *Software (compiler)*

- add nop → J operations dependent on I in the pipeline are delayed until I finishes
- instr scheduling (nop as fallback) → too dependent instructions are further distanced

- *Hardware*

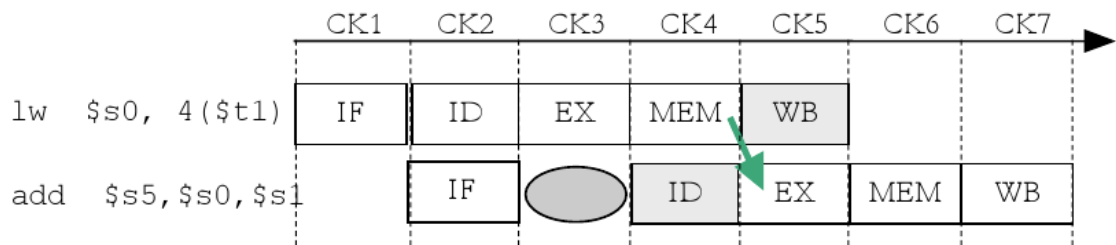
- add stall → hardware nop
- data forwarding/bypassing → results from pipeline stages are passed back to other pipeline stages (always possible except for LOAD/USE).

Need to add multiplexers at ALU inputs to fetch values from pipeline registers,
possible path are: EX/EX, MEM/EX, MEM/ID, MEM/MEM



Examples:

- LOAD/USE (only case where necessary to put stall/bubble even if forwarding)
add 1 stall: to permit MEM from instr I to complete before EX from instr J



- **WAW (Write After Write)**

Instr J tries to write operand before previous instr I writes it → writes in wrong order

I: sub **r1**, r4, r3

J: add **r1**, r2, r3

Solutions:

- *Hardware*

- use 5 stage MIPS: all instr take 5 stages and writes are always in stage 5 (WB)
- use another register to store the value

- **WAR (Write After Read)**

Instr J tries to write operand before previous instr I reads it → I read wrong value

I: sub r4, **r1**, r3

J: add **r1**, r2, r3

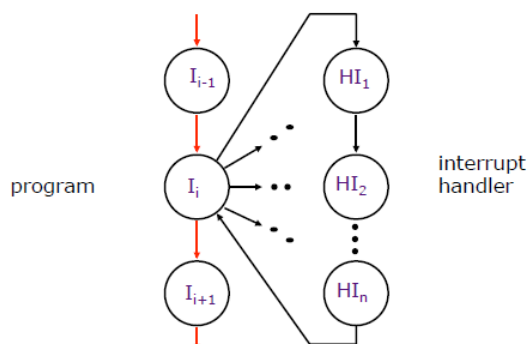
Solutions:

- *Hardware*

- use 5 stage MIPS: all instr takes 5 stages and read only in stage 2 (ID) and write only in stage 5 (WB)
- use another register to store the value if unrelated

Exception Handling

Interrupts



Alteration of the normal flow of control.

Usually unexpected external/internal event that needs to be processed by another sw.

Exceptions classes

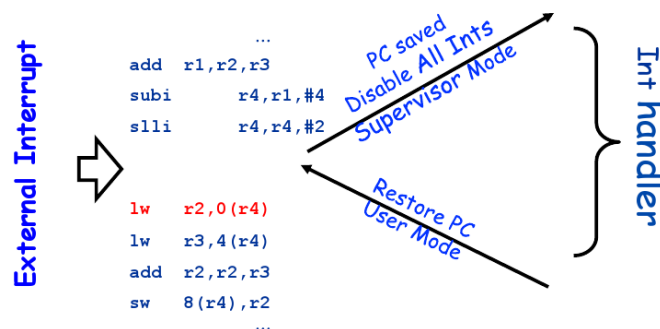
- **Synchronous vs Asynchronous:** async exceptions are caused by external devices from CPU and memory and can be handled after current instruction completion.
- **User requested vs Coerced:** User requested are predictable but treated as exceptions because share the same mechanisms.
Coerced are caused by hw event out of the control of the program.
- **User maskable vs user nonmaskable:** mask simply controls whether the hw responds to exceptions or not. Mask can be ignored while nonmasked cannot.
- **Within vs between instr:** exceptions that occur within instructions are usually sync (instr trigger the exception)² → stop+restart instr
exception between instr arise from catastrophic situations and always cause program termination
- **Resume vs terminate:** terminating event → sw stops after interrupt
Resuming event → sw continues after the interrupt

Asynchronous Interrupts

Interrupt handler invocation: I/O device asserts one of the prioritized interrupt request lines then CPU (in a transparent way):

1. stops current instruction and let finish all the instructions before
2. saves current instruction in EPC register
3. disable interrupts and transfers control to specific **interrupt handler** in kernel mode

Interrupt handler



1. Saves PC before enabling interrupts to allow nested interrupts:
 - moves PC to GPR (General Purpose Register)
 - masks further interrupts until PC can be saved
2. Reads a status register that indicates the cause of the interrupt
3. Uses RFE (Return From Exception jump) that:
 - enables interrupts
 - restores CPU to User mode
 - restores hw status and control state

Synchronous Interrupts

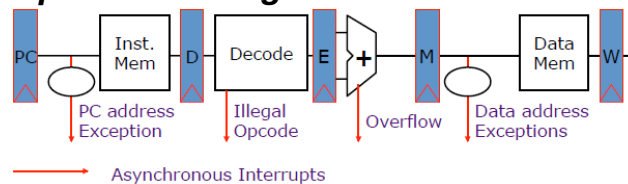
Caused by a particular instruction that cannot be completed and need to be restarted after exception has been handled. Instruction considered completed in case of a trap.

Precise interrupt: if all the previous instructions to the one that causes the interrupt have committed their state and the instruction that causes the interrupt and the next ones haven't. In sequential execution is easy, but not in out-of-order execution.

Execution can be restarted at interrupt point (easier to handle and restart).

Approximation to precise interrupts: handler must figure out how to find a precise PC where to restart the program (since hw has imprecise state during an interrupt) → Instr emulation in the pipeline.

5-stage pipeline exception handling



Solved by tagging instr in pipeline as “exception cause” and wait pipeline end to flag exception.

When instr get to WB: store PC → EPC, store interrupt vector addr → PC and turn all instr in the pipeline to nop.

How to get a precise PC:

- hold exception flags in pipeline until commit point (M stage)
- exceptions in earlier pipe stages override later exceptions for a given instruction
- inject external interrupts at commit point (override others)

If exceptions at commit: update Cause+EPC registers, kill all stages (fill all following stages with nops), inject handler PC into fetch stage.

Since exceptions are rare prediction mechanism is useless (predict that exceptions won't happen).

ILP – Instruction-level parallelism

Potential overlap of execution among unrelated instructions possible if:

- no Structural Hazards
- no RAW, WAR or WAW stalls
- no Control Stalls

Pipeline performance

$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural Stalls} + \text{Data Hazard Stalls} + \text{Control Stalls}$

- *Ideal pipeline CPI (=1):* measures max achievable performance
- *Structural Hazards:* HW cannot support this combination of instructions → need more hw resources
- *Data Hazards:* instruction depends on result of prior instructions still in pipeline → need forwarding, compiler scheduling
- *Control Hazards:* caused by delay between the fetching of instructions and decisions about

changes in control flow (branches, jumps, exceptions) → early evaluation and PC, delayed branch, prediction

increase pipeline → increase hazards, better branch prediction, more instruction parallelism, instruction bandwidth (not latency)

compilers → reduce cost of data and control hazards

CPIpipe reduced if right-hand term reduction

ILP vs Parallel Processing

2 ways to obtain parallel execution:

ILP (transparent to user)	Parallel Processing (nontransparent to user)
Overlap individual machine operations (add, mul, load, ...) → parallel execution (if no data dependencies). Goal: speed up execution	Having separate processors getting separate chunks of the program (processors programmed to do so) → multi-threading. Goal: speed up + quality up

Type of Data Hazards

Given $r_3 \leftarrow (r_1)op(r_2)$:

- RAW (data dependence)
 $r_5 \leftarrow (r_3)op(r_4)$
- WAR (anti dependence)
 $r_1 \leftarrow (r_4)op(r_5)$
- WAW (output dependence)
 $r_3 \leftarrow (r_6)op(r_7)$

Issues in Complex Pipeline Control

Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined FP units
- Memory systems with variable access time
- Multiple function and memory units
- Precise exception

Complex In-Order Pipeline

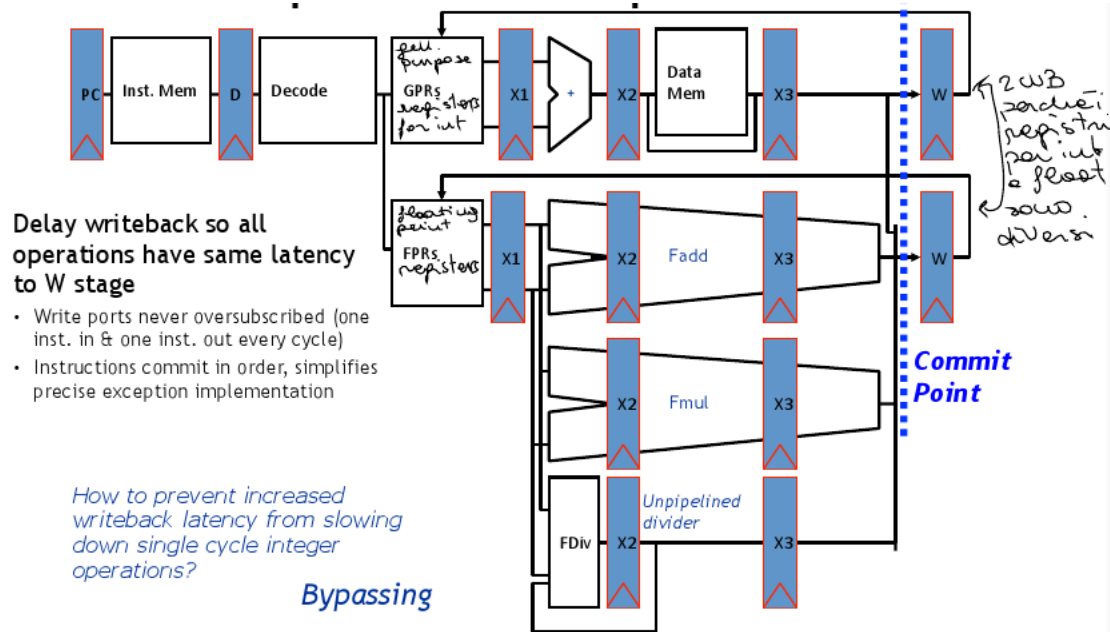
Issues:

- structural conflicts:
 - at EX stage if some FPU or memory unit is not pipelined and takes more than 1 cycle
 - at WB if FUs have different latencies → solution: apply “equalizing” to delay WB so all operations have same latencies and apply “bypassing” to prevent slow down of single cycle integer operations
- precise exceptions not possible → solution: in order instruction commit

fetch 2 instructions per cycle, issue both simultaneously if one is integer/memory and other is floating point

Easy to solve hazards by equalizing all operations length and bypassing, but doesn't yield max

performance.



Increase Performance

Can increase performance without “equalizing” and “bypassing”?

To reach higher performances: parallelism through instruction scheduling, but need to detect and solve dependencies.

Dependencies

Determining dependencies among instructions → achieve parallelism.

if dependency exists → no parallelism (overlapping) but in order execution.

- Dependencies → property of program
- Hazards → property of pipeline

3 dependencies types:

- **Name Dependencies (WAX)**

When 2 instructions use the same register or memory location (“name”), but no flow of data between the instructions associated with that name.

2 types of name dependencies, given instr “i” that precedes instr “j”:

- *Antidependence*: j writes a register/memory location that i reads → need to retain original order to prevent i reading the wrong value
- *Output dependence*: i and j writes the same register or memory location → last value must be j.

Solution: “register renaming”, change register/memory location. May be difficult since 2 different addr may be different but point to the same memory location (memory disambiguation).

Renaming can be done statically (compiler) or dinamically (hardware).

- **Data Dependencies**

Data/name dependency can potentially generate a data hazard (RAW, WAW or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline.

- **Control Dependencies**

Determines the ordering of instructions and it is preserved by 2 properties:

- in-order exec to ensure that an instr occurring before a branch is executed before the branch.
- detection of control hazards to ensure that an instruction is not executed until the branch direction is known.

“Control dependency” is not the critical property that must be preserved (even if control dependency is a simple way to control program order).

Program properties for correctness

Program correctness (preserved by maintaining data and control dependencies) achieved by:

- *exception behavior*: whatever the order of the instructions of the program, the exception order must remain the same.
- *data flow*: actual flow of data values among instr that produces the correctness.

Dynamic and static scheduling

2 strategies to support ILP scheduling:

	Static Scheduling (SW) boosted by parallel code optimization	Dynamic Scheduling (HW) boosted by static parallel code optimization
Done by:	compiler	Processor + parallel optimizing compiler
Processor receives:	dependency-free and optimized code for parallel execution	optimized code for parallel execution (but detects and solves dependencies on its own)
Used by:	VLIW and pipelined processors	pipelined and superscalar processors

Static Scheduling (software)

Basic blocks (small piece of independent code, except for entry and exit points) are reordered to enhance parallelism.

Cons:

- unpredictable branches
- variable memory latency
- code size explosion
- compiler complexity

Keep pipeline full or use all FU in each cycle as much as possible to reach better ILP and therefore higher parallel speedups.

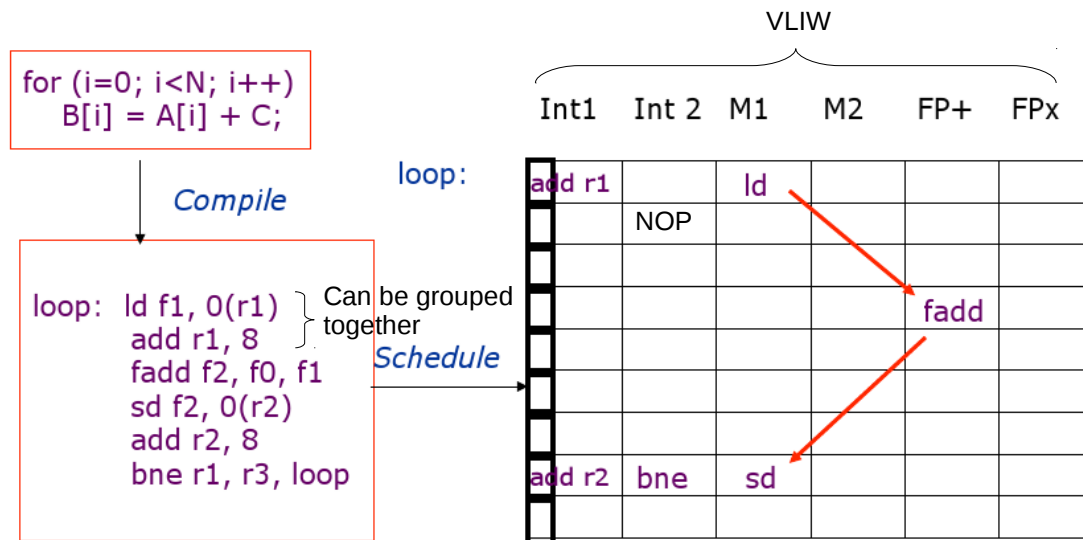
Methods:

- Simple code motion (no scheduling)
- Global code scheduling (on basic blocks)

- Loop unrolling
- Sw pipeline
- Trace Scheduling
- Superblock Scheduling
- Hyperblock Sheduling

Loop unrolling

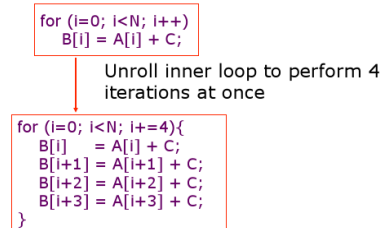
Normal Loop Execution (FLOPS/cycle = 1 fadd / 8 cycles = 0.125)



Loop Unrolling (FLOPS/cycle = 4 fadds / 11 cycles = 0.36)

To unroll a loop must make assumption on the number of times the loop will be repeated.

loop unrolled and instruction destined to same FU are grouped together and executed in-order.



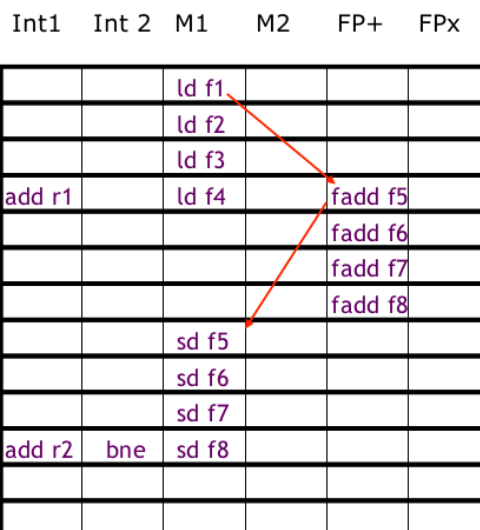
Example: not greater than 4

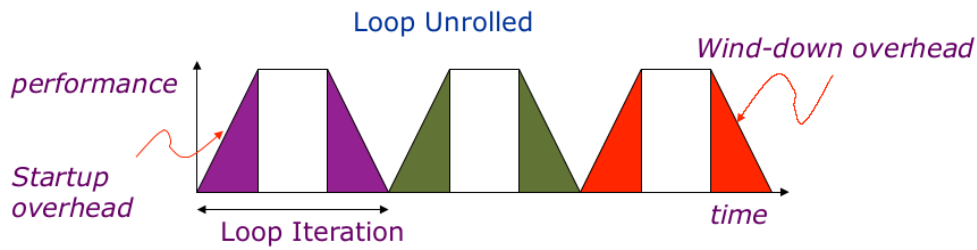
Unroll 4 ways

loop: ld f1, 0(r1)
ld f2, 8(r1)
ld f3, 16(r1)
ld f4, 24(r1)
add r1, 32
fadd f5, f0, f1
fadd f6, f0, f2
fadd f7, f0, f3
fadd f8, f0, f4
sd f5, 0(r2)
sd f6, 8(r2)
sd f7, 16(r2)
sd f8, 24(r2)
add r2, 32
bne r1, r3, loop

loop:

Schedule





Disadvantages:

- increase number of registers
- increase code size
- check for dependencies between iterations at compile time

Software Pipelining (Exploit VLIW in unrolled loops) (4 fadds / 4 cycles = 1)

Overlaps green and violet instructions.

Unroll 4 ways first

```

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
  
```

prolog

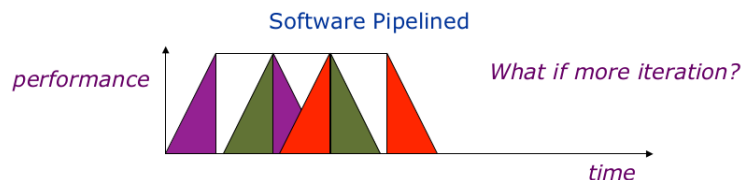
iterate

epilog

How many FLOPS/cycle?
4 fadds / 4 cycles = 1

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
		ld f3	sd f7	fadd f7	
		ld f4	sd f8	fadd f8	
		sd f5	fadd f5		
		sd f6	fadd f6		
		sd f7	fadd f7		
		sd f8	fadd f8		
		sd f5			

Startup/wind-down costs only once per loop, not once per iteration.

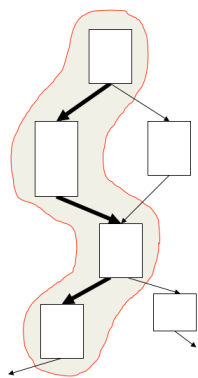


Trace Scheduling

Problem: if no loops, it's difficult to find ILP in individual basic blocks

Trace: loop free sequence of basic blocks embedded in the control flow graph (Fisher).

Execution path can be taken multiple times (they differ in execution frequency) and with different inputs.



Profiling feedback or compiler heuristics to find common branch paths.

Whole trace is scheduled at once. Add fixup code to jump out of the trace if necessary.

Pros:

- Extract more ILP
- Increase machine fetch bandwidth by storing logically consecutive blocks in physically contiguous cache location (load more basic block in one cycle)
- implemented via sw (based on static profiled data) or hw

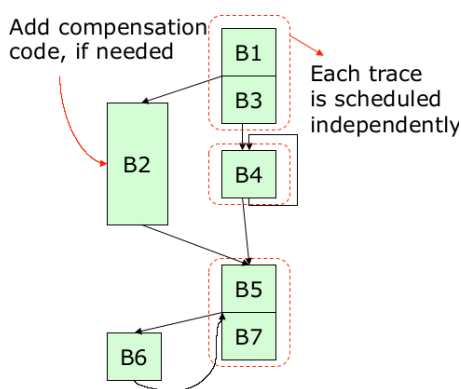
Cons:

- **Loops:** trace scheduling cannot work on loops → loop unrolling
 - extra code
 - performance loss
 - closing iterations

Compensation codes: needed for:

- side entry points (difficult)
- side exit points

Example:



Possible traces are:

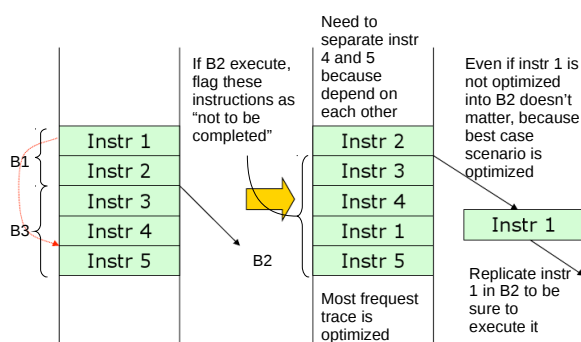
- B1, B3
- B4 (single execution of the loop)
- B5, B7
- B1, B2
- B1, B2, B5, B6, B7
- B1, B2, B5, B7

Assume: B1, B3, B4, B5, B7 most frequently executed path therefore traces are:

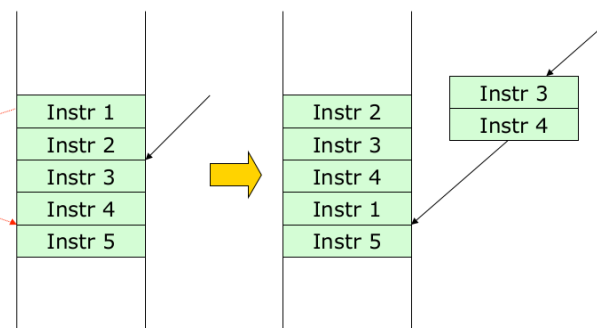
- B1, B3
- B4
- B5, B7

Then compensation examples can be:

Move instr below a **side exit** (simple)
exiting branch
costs 1 more instruction



Move instr below a **side entry** (complex)
incoming branch
costs 2 more instructions



Rules:

- dataflow must not change (semantic)
- Exception behavior must not change

Dataflow can be guaranteed by:

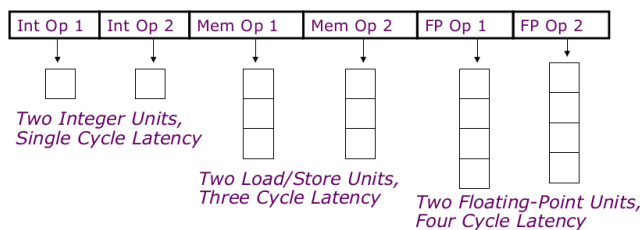
- data dependency
- control dependency, can be solved by:
 - predicate instructions (hyperblock scheduling) → remove branching
 - speculative instructions (speculative scheduling) → speculatively move an instruction before branch

VLIW (Very Long Instructions Word)

Processor can initiate multiple operations per cycle (completely specified by compiler, not like superscalar).

- Low hw complexity
- explicit parallelism
- single control flow
- (no scheduling hw)

VLIW instructions



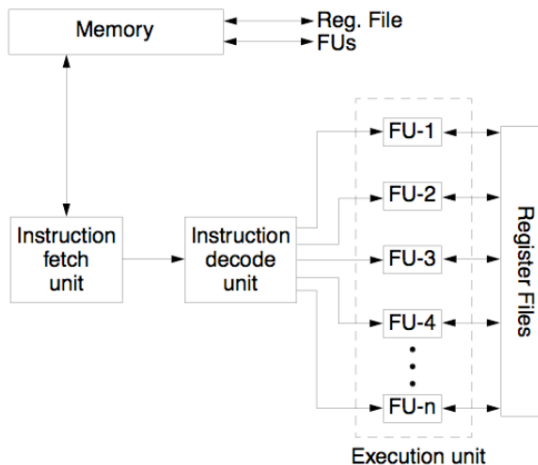
Independent instructions are grouped into blocks, with each instruction bonded to a different FU.

Constant operation latencies are specified.

Must assure:

- no data use before it's ready
- no need for RAW checks

VLIW machine



1. **Instruction fetch unit:** fetches a block of instructions at a time
2. **Instruction decode unit:** block of instructions is processed and redirected to the proper FU
3. **Execution unit:** all FU works in parallel starting at the same time

VLIW compiler

- Schedules to max parallel execution
- Map instr over the machine functional units
- Guarantees intra-instruction parallelism
- Avoids data hazards (use of “nop”)

goal: minimize total execution time for the program

Pros and Cons

Pros:

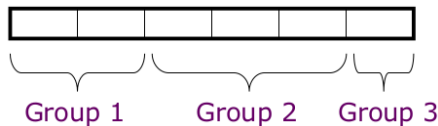
- Simple hw
- good compilers effectively detect parallelism

Cons:

- Huge number of registers to keep active the FUs
- Large data transport capacity:
FUs ↔ register files ↔ memory
- High bandwidth between i-cache and fetch unit
- Large code size

Loop execution → sw pipelining

VLIW instruction encoding

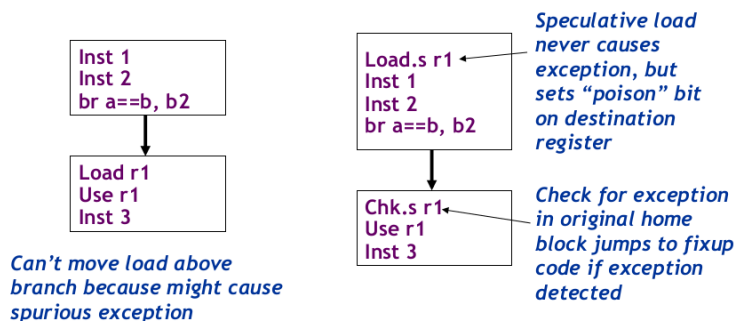


Schemes to reduce effect of unused fields

Problems

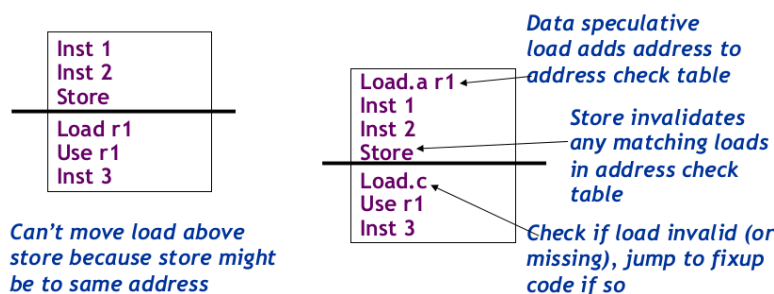
- **Branches restrict compiler code motion**

solution: speculative operations that don't cause exceptions. Good for long latency operations that are started early.



- **Possible memory hazards limit code scheduling**

solution: hw to check pointer hazards



- **Mispredicated branches limit ILP**

solution: eliminate hard to predict branches with "predicated execution".

Predicate registers may be true or false, instr with false predicate register become nop.

Problems

Problems with classic VLIW:

- Object-code compatibility → need to recompile for each machine
- Object-code sizes → instr padding + loop unrolling are bad for performances
- Scheduling variable latency memory operations → caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities → need for profiling of applications
- Scheduling for statically unpredictable branches → optimal schedule varies with branch path

Dynamic Scheduling (hardware)

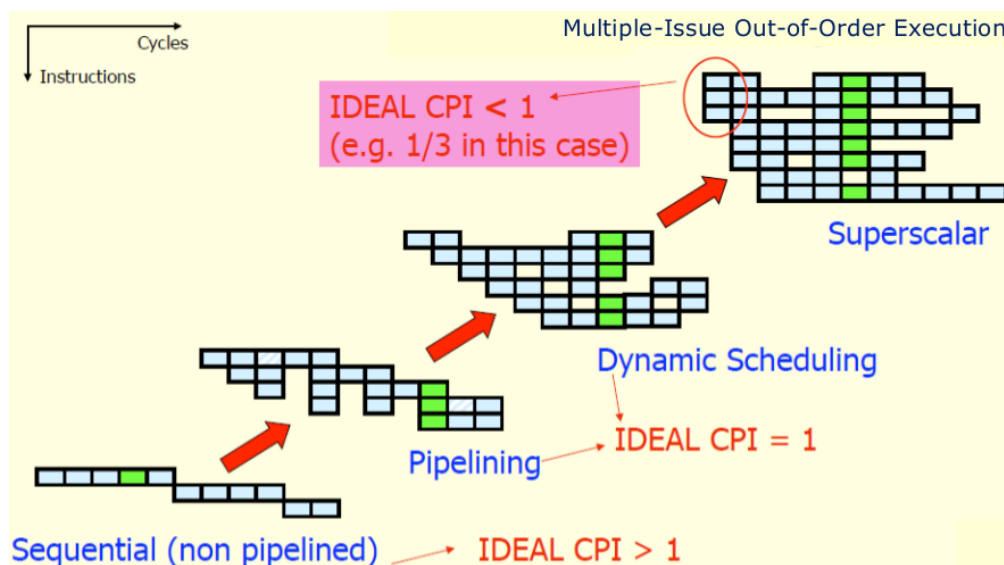
Hw reorders instruction execution to reduce pipeline stalls while maintaining data flow and exception behavior. Needs hw scheduler and more IF units.

- | | | | |
|-------|---|-------|--|
| Pros: | <ul style="list-style-type: none">• handles dependencies unknown at compile time• simplifies compiler complexity• compiled code can run efficiently on multiple pipelines | Cons: | <ul style="list-style-type: none">• complex hw• power consumption• imprecise exception (risk, need more hw to handle it) |
|-------|---|-------|--|

Instructions are:

- fetched (to verify data dependencies) and issued in-order (also more than one at a time)
 - executed out-of-order as soon as the operands are available
- but: introduction of WAR/WAW and out-of-order completion (unless re-order buffer)

Several steps to exploit more ILP:

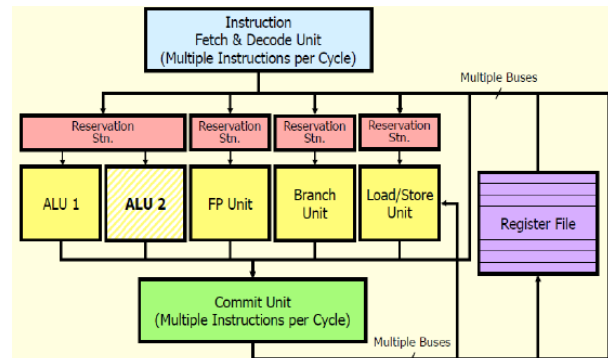


Superscalar execution

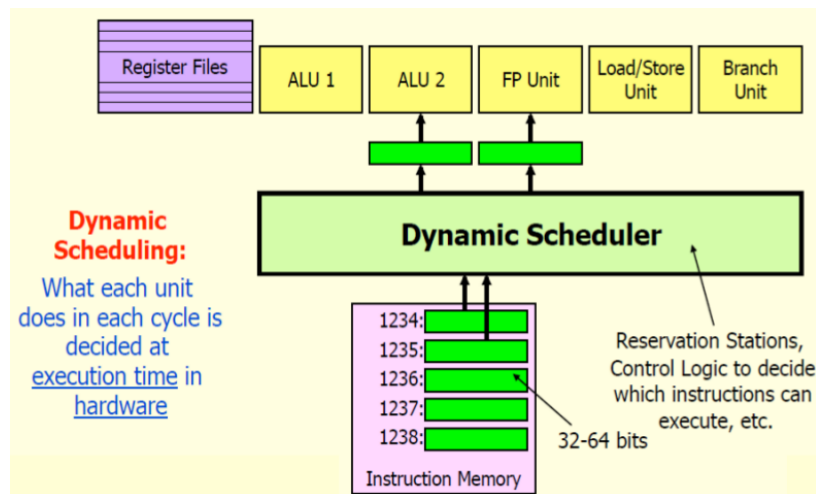
Fetching multiple instruction at each clock cycle.

Requirements:

- Fetch Unit: bigger instruction cache and manages more requests at the same time.
- Decide on data control dependencies: dynamic scheduling and dynamic branch prediction.



Dynamic Scheduler



At every cycle, CPU decides which instruction can begin execution.

Need to check all (limited) fetched instructions with all in-flight instructions to see which are independent and therefore can start execution.

Basic data structure for instruction issuing

Can issue instruction "i"?

Name	BUSY	OP	DEST	SRC1	SRC2
Int	FU available?		RAW: check for i sources WAW: check for i dest	WAR: check i dest	
Mem					
...					

Check also for structural conflict at WB stage.

If no hazard detected add I to table, remove instruction after WB.

Scoreboard

Dynamic scheduler [...]

Pro and cons

Pros:

- handle cases where dependencies not clear at compile time

Cons:

- energy consumption

- (only in runtime)
- simplifies compiler complexities
- allow compiled code to run efficiently on a different pipeline
- hw complexity

Summary

Technique	Reduces
Dynamic scheduling	Data hazard stalls
Dynamic branch prediction	Control hazards stalls
Multiple issue	CPI_ideal
Speculation	Data and control stalls

Scoreboard basic scheme

Referring to “basic data structure for instruction issuing”

Name	BUSY	OP	Dest	SRC1	SRC2
Int	FU available?		RAW: check for i sources NO WAW: register name at most once in DEST	NO WAR: no need to keep src1 and src2	

- WP[reg#] a bit-vector to record the registers for which writes are pending → bits set by the issue stage and cleared by WB stage. Each pipeline stage in FU carries the DEST field and a flag to indicate if it's valid.

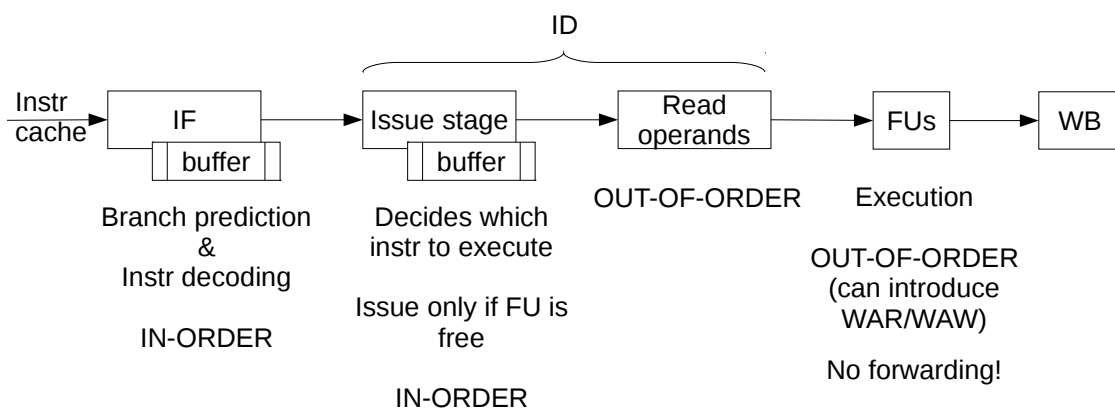
Exception handling

Must preserve behavior as in-order execution → ensure that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed.

Imprecise exception: if processor state when an exception is raised does not look exactly as if the instructions were executed in-order. Difficult to restart the execution after handling.

- Pipeline has completed instructions that are later wrt the instruction raising the exception
- Pipeline has not yet completed some instructions that are earlier in the program order than the instruction causing the exception

4 stages of Scoreboard Control



1. **Issue:** decode instr and check for structural hazards. Issue in-order (hazard checking).
If (FU free & DEST free (no other ongoing instruction needs it))
then scoreboard issues instruction to the FU and updates the data structure
else WAW exists, so the instruction stalls and issuing halted until hazards are solved
2. **Read Operands:** wait until no data hazards, then read operands.
if (no issued instruction writes in SRC OR FU is writing its value in the register)
then SRC operand is free. Scoreboard tells FU to proceed to read operands from registers and begin execution. RAW solved in this stage, out-of-order execution is possible.

Issue iff instruction passes stages 1 and 2

3. **Execution:** operate on operands. FU begins execution upon receiving operands. When finished it notifies the Scoreboard.
FU characterized by:
 - *latency*: effective time to complete one operation
 - *initiation interval*: # of cycles between issuing 2 operations to the same FU
4. **WB:** write results and finish execution. Once Scoreboard is aware that the FU has completed execution, it checks for WAR hazards.
If (WAR hazards) *then* stall instruction *else* write results

Scoreboard Structure

1. Instruction status

<i>Instruction status:</i>				<i>Read Exec Write</i>		
Instruction		<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Oper</i>	<i>Comp Result</i>
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

2. FU status, indicates state of local FU

Busy – whether FU is busy or not F_j, F_k – source register numbers

Op – operation to perform in the unit Q_j, Q_k – FU producing SRC registers

F_i – destination register R_j, R_k – flags indicating when F_j, F_k are ready

<i>Functional unit status:</i>		<i>dest</i>	<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>		
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>	<i>Rk</i>
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

3. **Register result status**, indicates which FU will write each register.

Blank if no pending instruction will write that register.

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
<i>FU</i>									

Scoreboard pipeline control

Instruction Status	Wait until	Bookeeping
Issue	$!busy(FU) \wedge !result(DEST)$	$busy(FU);$ $op(FU) \leftarrow op;$ $F_i(FU) \leftarrow D;$ $F_j(FU) \leftarrow S1;$ $F_k(FU) \leftarrow S2;$ $Q_j(FU) \leftarrow result(S1);$ $Q_k(FU) \leftarrow result(S2);$ $R_j \leftarrow !Q_j;$ $R_k \leftarrow !Q_k;$ $result(D) \leftarrow FU;$
Read operations	$R_j \wedge R_k$	$R_j \leftarrow No; R_k \leftarrow No$
Execution complete	$free(FU)$	
Write results	$\forall f \left(\begin{array}{l} (F_j(f) \neq F_i(FU) \vee R_j(f) = No) \wedge \\ (F_k(f) \neq F_i(FU) \vee R_k(f) = No) \end{array} \right)$	$\forall f (Q_j(f) \Rightarrow R_j(f) \leftarrow Yes);$ $\forall f (Q_k(f) \Rightarrow R_k(f) \leftarrow Yes);$ $result(F_i(FU)) \leftarrow 0;$ $busy(FU) \leftarrow No;$

How To

1. Instruction Status

get next instruction I (in-order)

- if (FU (Integer for Load/Store ops) of I is free)
then issue: write current clock in Issue column
else stall issuing

continue execution of previously issued instruction if parameters are ready: write current clock in Read Oper, Exec Comp or Write Result (depending on the case).

Write result permitted only if no other previous instruction must read result from same location (WAR)

2. Functional Unit Status

if (I issued)

then update corresponding FU:

Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
-------------	-----------	------------	------------	------------	------------	------------	------------	------------

Yes	Operation type	DEST	SRC1	SRC2	FU SRC1	FU SRC2	F_j ready flag	F_k ready flag
-----	----------------	------	------	------	---------	---------	----------------	----------------

If previous instruction has reached Read Op in previous cycle then negate corresponding R_j and R_k flags

If previous instruction has reached Write Result then remove instruction from FU table.

If previous instruction has reached Read Op then start clock countdown for execution completion

3. Register Result Status

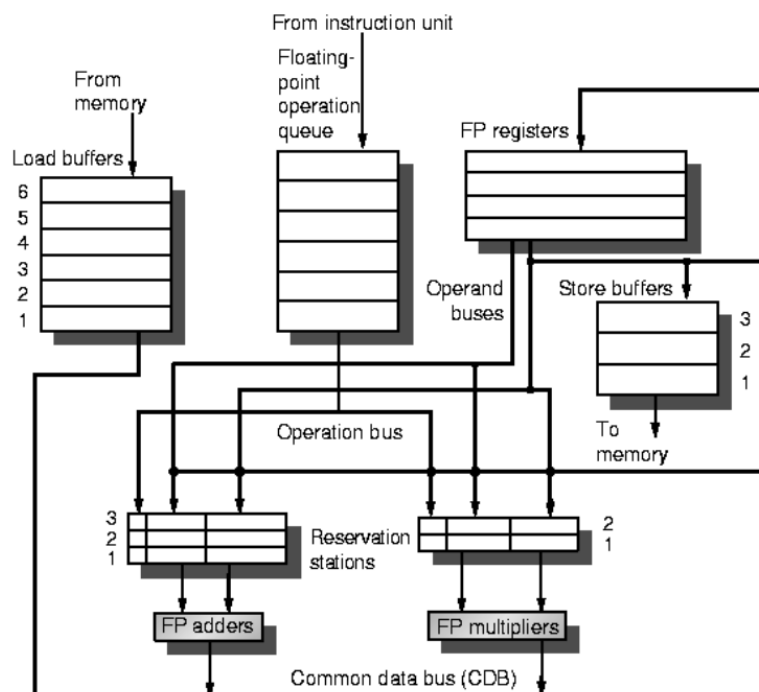
if (I issued) then reserve result register: write to corresponding I the reserved FU

Tomasulo Algorithm

Characteristics

- Control and buffers distributed with FU (reservation stations with pending operands)
- Register renaming: register in instructions are replaced by values or pointers to RS (reservation stations).
- No WAR, WAW hazards.
- More RS than registers → more optimizations
- Common Data Bus: FU results are broadcasted directly to all FU
- LOAD/STORE treated as FU with RS as well

Architecture



Components

- RS (Reservation Station):**
 - Tag:** identify RS
 - OP:** operation to perform on the component
 - V_j, V_k:** value of SRC operands

only 1 v-field or q-field is valid for

- Q_j, Q_k : pointer to RS that produce V_j, V_k each operand.
if $=0$ then SRC already available in V_j or V_k
- *Busy*: indicates whether RS is busy
- **Load buffer**: takes a second execution step to access memory, then go to WB to send the value from memory to RF and/or RS.
Executed out-of-order with previous store: assume address computed in program order.
When Load has been computed, it can be compared with A fields in active Store Buffers:
if match → Load is not sent to Load Buffer until conflicting Store completes.
 - A: address
 - *Busy*
- **Store buffer**: complete execution in WB stage.
 - A: address
 - V:
 - Q: number of RS producing the result to be stored in RF or Store Buffer
if $=0$ then no active instructions producing the result

Stages

Issue

Get instruction I from queue. If I is an FP op → check if RS available (structural hazards)

- Rename Registers
- WAR resolution: If I writes R_x and R_x was already read by already issued instruction K then K already knows the value of R_x , so RF can be linked to I.
- RAW resolution: link RF to I (because of in-order issue).

Execution

If both operands ready then execute, else check CDB for results.

- RAW: avoided by waiting for available operands

LOAD/STORE: kept in program order through effective address calculation, helps preventing hazards through memory.

1. compute effective address place it in Load or Store buffer
2. loads in Load buffer and executes as soon as memory unit is available.

Store buffer waits for the value to be stored before being sent to memory unit

Preserve exception:

No instruction can start execution until all branches preceding it in program order have completed.

If branch prediction used → CPU must know prediction correctness before beginning execution or following instructions.

WB

If result available → write it on CDB and from there into RF and into all RS waiting for this result.

Store also writes data to memory during this stage. Mark RS as available.

All writes happen in WB.

Details

- Load/Store go through FU for address computation before executing Load/Store
- Load/Store can be done in different order if they access different memory areas, otherwise:

- WAR: interchange load → store
- RAW: interchange store → load

to detect hazards: memory addresses associated with any earlier memory operation must have been computed by CPU

Drawbacks

- Complexity: large amount of hw
- Many associative stores (CDB) limit performances

How To

For loops first load instructions from n unrolled cycles and then executes them with 1 clock cycle of difference.

1. Instruction Status

get next instruction I (in-order)

1. if (I is LOAD && load unit available)

then issue: first available load unit: Busy yes, Address $j \text{ (offset)} + k \text{ (base)}$ and write current clock in issue column

else if (I is FU && FU available) then issue: write current clock in issue column

else stall issuing

continue execution of previously issued instruction if parameters are ready: write current clock in Read Oper, Exec Comp or Write Result (depending on the case).

If previous load instruction has reached Write Result then remove instruction from load table.

2. Reservation Stations

if (I issued)

then update corresponding FU:

Busy	Op	V _j	V _k	Q _j	Q _k
Yes	Operation type	SRC1	SRC2	FU/LOAD SRC1	FU/LOAD SRC2

If previous instruction has reached Write Result then remove instruction from FU table.

If previous instruction has reached Read Op then start clock countdown for execution completion

3. Register Result Status

if (I issued) then reserve result register: write to corresponding I the reserved FU or Load unit.

If previous instruction has reached Write result status substitute with memory location (also update Reservation Station)

Hardware Based Speculation

Combines 3 ideas:

- **Dynamic Branch Prediction:** choose which instruction to execute
- **Dynamic Scheduling:**
 - out-of-order execution
 - in-order commit to prevent any irrevocable actions (register updates/exceptions)
also bypassing is supported
- **Speculation:** exec instr before control dependencies are solved (before branch outcome is known)

when instruction becomes certain (no more speculative) → allow to update register file or memory. Need for specialized hw.

Similar to precise interrupts: take best shot at predicting a branch, but if wrong rollback and execute to point at which prediction was wrong.

Reorder Buffer

Goal: reorder out-of-order instruction to get in-order commit and buffer the results.

When instruction complete → results placed in ROB

- tag: provides tag to results instead of RS
- supplies other operations with needed operands

Queue processing: FIFO (as issued) with a circular buffer with head/tail pointers:

- head: commit of instructions in registers
- tail: allocation of new instructions

RO buffer main entries:

V	I	OPCODE	P	TAG	SRC1	P	TAG	SRC2	P	REG	RESULT	EXCEPT
---	---	--------	---	-----	------	---	-----	------	---	-----	--------	--------

- Tag: given by index in ROB
- SRC1, SRC2: non busy src operands on dispatch reads from RF
- p: presence, if empty indicates that instruction has/will produce value. If trap, reset all P=1
- v: valid on dispatch

- completed instructions cannot retire as long as they are on speculative status.

Exceptions are handled by not recognizing exception until instruction that caused it is ready to commit in ROB. Can handle precise exceptions.

The diagram illustrates a processor pipeline with an exception handler. The pipeline stages are Fetch (In-order), Decode (In-order), Reorder Buffer (Out-of-order), Execute (Out-of-order), and Commit (In-order). An exception handler block labeled "Exception?" is connected to the Reorder Buffer, Execute, and Commit stages. Red arrows labeled "Kill" point from the exception handler back to the Fetch, Decode, and Reorder Buffer stages. A red arrow labeled "Inject handler PC" points from the exception handler back to the Fetch stage.

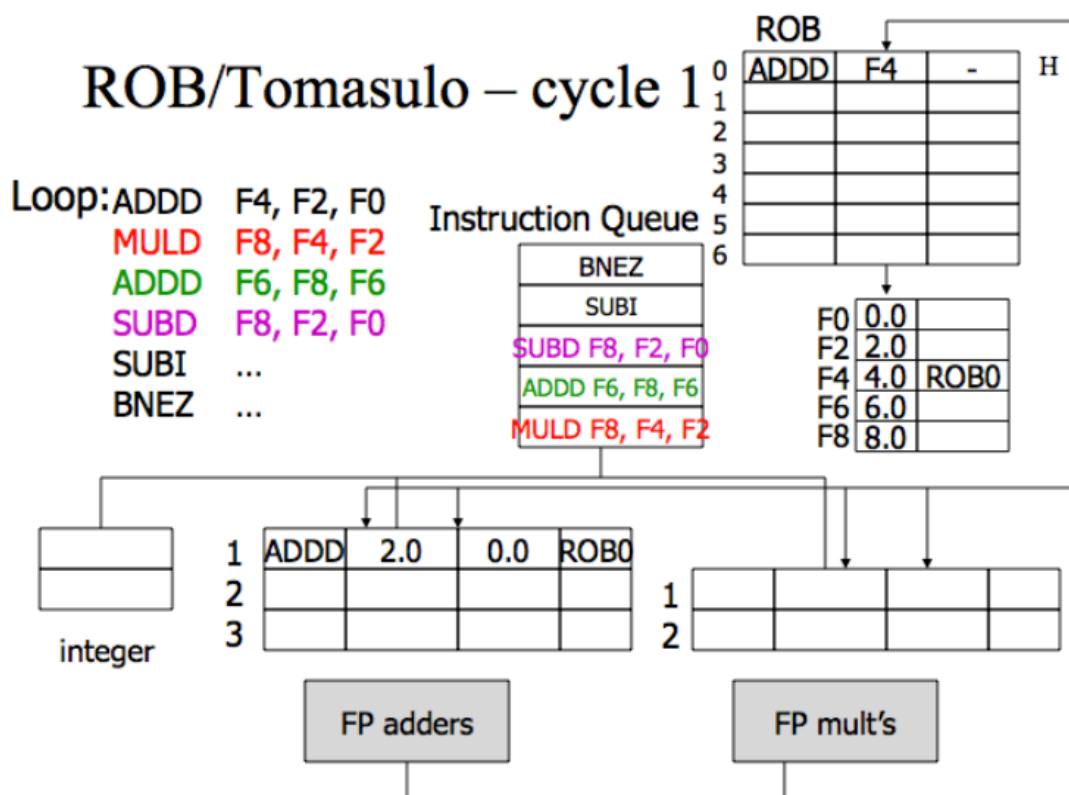
ID does register renaming and adds instructions to the issue-stage instruction reorder buffer (ROB)
→ NO WAR, WAW.

The diagram illustrates the RISC-V architecture with the following components and data flow:

- From instruction unit:** Provides input to the **Reorder buffer** and the **Instruction queue**.
- Reorder buffer:** Buffers instructions before they reach the **Instruction queue**. It receives a red arrow from the right, indicating a write operation.
- Instruction queue:** A queue of instructions waiting to be executed. It feeds into the **Address unit** for load/store operations and the **Floating-point operations** unit.
- Address unit:** Handles address generation for load/store operations. It feeds into the **Load buffers** and the **Memory unit**.
- Load buffers:** Buffers for load operations, feeding into the **Memory unit**.
- Memory unit:** Manages memory access. It receives **Store data** and **Store address** from the **Address unit** and provides **Load data** back to the **Address unit**.
- Floating-point operations:** A unit that handles floating-point arithmetic. It receives input from the **Instruction queue** and the **Operation bus**.
- Operation bus:** A central bus that coordinates data flow between the **FP registers**, **Reservation stations**, **FP multipliers**, and the **Common data bus (CDB)**.
- FP registers:** Floating-point registers that store data. They receive **Reg #** and **Data** from the **Operation bus** and provide **Operand buses** to the **FP multipliers**.
- Reservation stations:** Units that hold instructions waiting for operands. They are numbered 1, 2, and 3. They feed into the **FP multipliers**.
- FP multipliers:** Floating-point multiplier units that perform multiplication. They feed into the **Common data bus (CDB)**.
- Common data bus (CDB):** A shared bus that connects all major components of the processor.

1. **Issue** – get instruction from FP op queue
If (RS and ROB slot are free)
then issue instr, send operands and reorder ROB number destination

2. **Execution** – operate on operands (EX)
 - if (both operands are ready || in RS) then execute (check for RAW)
 - else check CDB for results
3. **Write results** – finish execution (WB)
 - write on CDB to all awaiting FU and reorder ROB.
 - Mark RS available.
4. **Commit** – update register with reorder result
 - if (instr at head of ROB && result is present)
 - then update register with result (or store to mem) and remove instr from ROB
 - mispredicted branch flushes ROB
 - 3 possible commit available:
 1. **normal commit**: instr reaches head of ROB (result present in buffer), result is present in the buffer → store result in the register and remove instr from ROB
 2. **store commit**: same as normal commit but store results in memory
 3. **instr is a branch with incorrect prediction**:
 - wrong speculation → flush ROB (graduation), restart exec at correct branch successor.

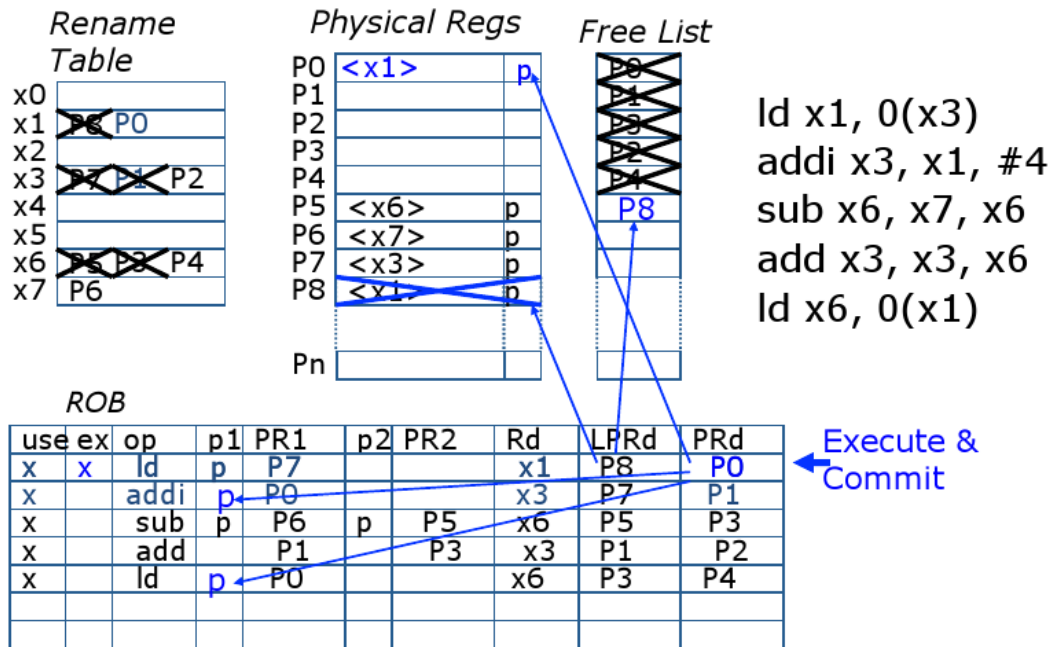


Register Renaming

Explicit Register Renaming: physical register file that is larger than the number of registers specified by the ISA (while Implicit Register Renaming: user registers renamed to RS tags).

Allocates a new physical destination register for every instruction that writes (NO WAR, WAW).

Interrupts: undo table mappings after commit:



Scoreboard + Explicit Renaming

No need to check for WAR or WAW hazards.

- Issue:** decode instr and check for structural hazards and allocate new physical register for results:
 - instr issued in program order (hazard checking)
 - no issue if no free PR
 - no issue if structural hazard
- Read operands:** wait until no hazards, read operands
RAW solved at this stage (wait for instr for WB)
- Execution:** operate on operands
FU executes upon receiving operands and then notifies scoreboard when results are ready
- Write results:** finish execution

ILP Limits

Ideal machine

- register renaming INFINITE → WAW and WAR avoided
- branch prediction PERFECT → no misprediction
- jump prediction PERFECT → all jumps correctly predicted
- memory-address alias analysis KNOWN → store can be moved before load if address NE
- 1 cycle latency for all instructions and unlimited number of instr issued per clock cycle

all instr in the window should be kept in the cpu

Limits on window size

Dynamic analysis necessary to approach perfect branch prediction (not possible at compile time).
Perfect dynamic scheduled CPU should:

- look arbitrarily far ahead to find best set of instructions to issue and make perfect predictions
- rename all registers (no WAW/WAR)
- renaming of possible data dependencies
- handle memory dependencies
- enough FUs to allow all ready instr to issue

Window size depends on the number of comparisons necessary to determine RAW dependencies.

3 models of memory alias

Need to check addr of each load and each store. 3 possible memory alias analysis:

- Global/stack perfect: assumes perfect prediction for all global+stack references (compiler), conflict on all heap references (dynamic values, compiler useless)
- inspection: accesses screened at compile time to check for interferences.
- None: all memory references are assumed to conflict

Other form of parallelism that can help are: Multi-core and SIMD

Multithreading and multicore processors

Why multiprocessing:

- low efficiency in silicon and energy
- interest in high-end servers
- growth for data intensive applications
- no single source of problems (memory conflicts, hazards, branch mispredictions, ...)

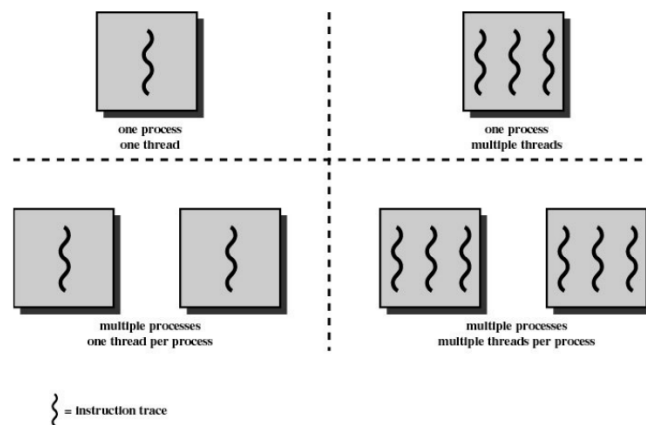
2 types of explicit parallelism:

- thread level: instruction stream with own pc and data, may be independent program or a process part of a parallel program.
- data level: perform identical operations on a lot of data

Parallel programming

Explicit parallelism implies structuring applications into concurrent and communicating tasks.

OS supports different types of tasks: processes and threads



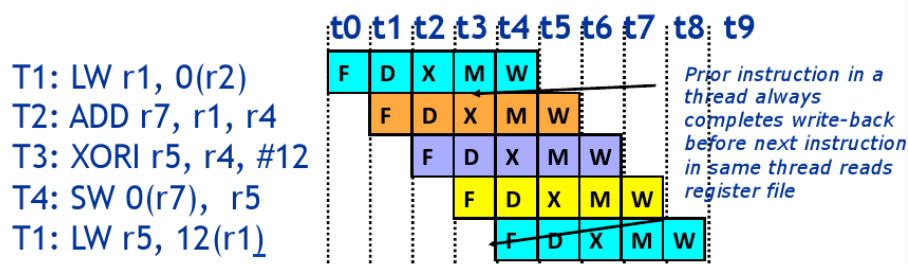
Multi-threading

Sharing of functional units of 1 processor via overlapping:

- maintain the copy of the state of each thread (PC, RF, page table, ...)
- memory shared through the virtual memory mechanisms
- fast thread switching

No dependency between instr guaranteed by execution of instruction from different program threads on the same pipeline.

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe



TLP: multiprocessor with multiple independent threads operating at once in parallel.

When to switch thread?

- **Fine grained:** switch thread at each instruction (execution of instructions from multiple threads interleaved) skipping stalled threads (good), but individual thread slowed down because ready thread must wait for other threads to complete (bad).
 - *Fixed interleave:* each of the N threads executes one instr every N cycles, if thread not ready insert bubble.
 - *SW controlled interleave:* OS allocates S pipeline slots amongst N threads. HW performs fixed interleave over S slots executing whichever thread is in that slot.
 - *HW controlled thread scheduling:* HW keeps track of which threads are ready to go. Next thread chosen on HW priority scheme.
- **Coarse grained:** execute another thread when thread in execution is “heavily” stalled (eg:

cache miss).

Advantage: in normal conditions the single thread is not slowed down.

Disadvantage: for short stalls it does not reduce the throughput loss. Better for costly stalls when pipeline refill \ll stall time.

- **Simultaneous MultiThreading (SMT)**: exploits ILP and TLP with superscalar processor resources. Fine grained multithreading + multiple issue dynamically scheduled processor.

Register renaming + dynamic scheduling allows for multiple execution without considering dependencies.

Fetch unit in a SMT can: fetch multiple instr threads at once, choose which threads to fetch (the one with fewer unresolved branches, fewer load misses, fewer instructions in queue)



Multithreading reduces execution time by more than it increases avg power. ILP instead (speculation, predicated execution, ...) don't consider power execution.

Multicore

Why multicore?

- Deep pipeline are problematic (heat dissipation, speed transmission over wires, complex)
- many application are multi-threaded
- difficult to improve single core performance and clock

Multicore and SMT

- Processors can use SMT
 - N threads: 2, 4, 8
 - memory hierarchy:
 - multithreading only → all caches shared
 - multicore → cache L1 private, cache L2 private/shared, memory shared
- need bigger caches to support multiple threads at once

Parallel Architecture: SIMD and vector architectures

Parallel Architectures

Parallel computer is a collection of processing elements that cooperates and communicate to solve large problems fast.

Goal: replicate processor to add performance (instead of design faster processor) → extend traditional computer with a **communication architecture**

Parallelism in applications

- **Data-level parallelism (DLP):** many data items that can be operated at the same time
- **Task-level parallelism (TLP):** tasks of work largely independent from each other

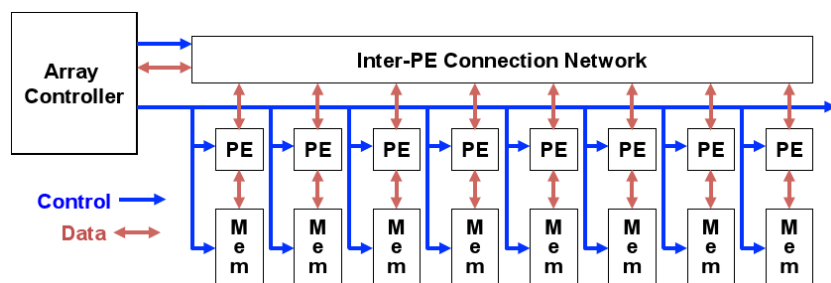
Hardware parallelism

- **ILP:** DLP at modest level with compiler techniques (pipelining) and at medium level with speculation
- **GPU:** DLP by applying single instruction to a collection of data in parallel
- **Thread level parallelism:** DLP/TLP parallelism in a tightly coupled HW with interaction among threads
- **Request-level parallelism:** TLP among largely decoupled tasks specified by the programmer or the OS

Multiprocessor classification

- **Single Instruction Single Data (SISD):** uniprocessor system
- **Multiple Instruction Single Data (MISD):** does not exist
- **Single Instruction Multiple Data (SIMD):** Simple programming model, low overhead, flexibility, custom integrated circuits. Good for data level parallelism.
- **Multiple Instruction Multiple Data (MIMD):** scalable, fault tolerant, off-the shelf micros

SIMD

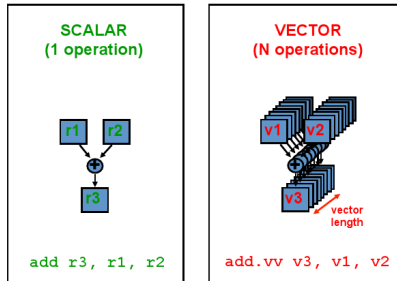


- one controller for the whole array
- one copy of the program in the storage (sequential program)
- all computations fully synced

3 SIMD variations:

- **x86**: MMX, SSE, AVX
- **GPU**: heterogeneous architecture (needs processor and system memory in addition)
- **vector architectures**: read sets of data elements into vector registers → operate on data → disperse results back into memory.

Vector Architectures



Can adapt to different sizes. Supercomputers are vector machines specialized at doing few operations faster than other computers on big amount of data.

VMIPS architecture

Vector registers: 8 registers of 64 elements @ 64bit

- **RF**: 16 read ports and 8 write ports

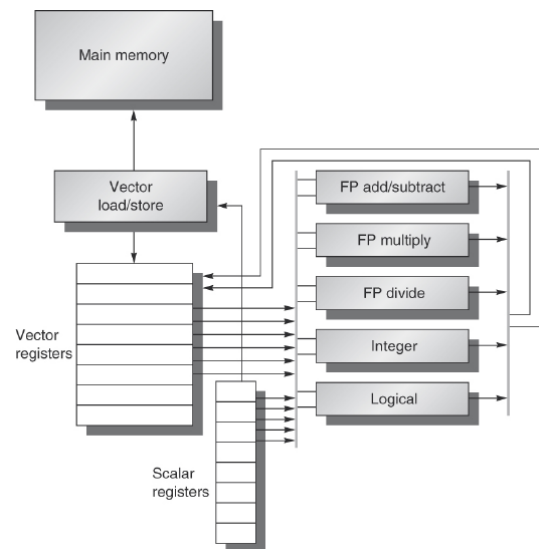
Vector FU: fully pipelined to start a new operation at each cycle

Vector load/store unit: fully pipelined 1 word per clock cycle after initial latency

Scalar registers: 32 general-purpose registers, 32 floating-point registers

arithmetic operations:

- deep pipeline: fast clock to execute element ops
- no hazards since vector elements are independent



Advantages over scalar:

- Vector operations chaining: vector op can start as soon as one element of the vector is available. Results from 1° FU are then forwarded to the 2° FU which can start the computation.

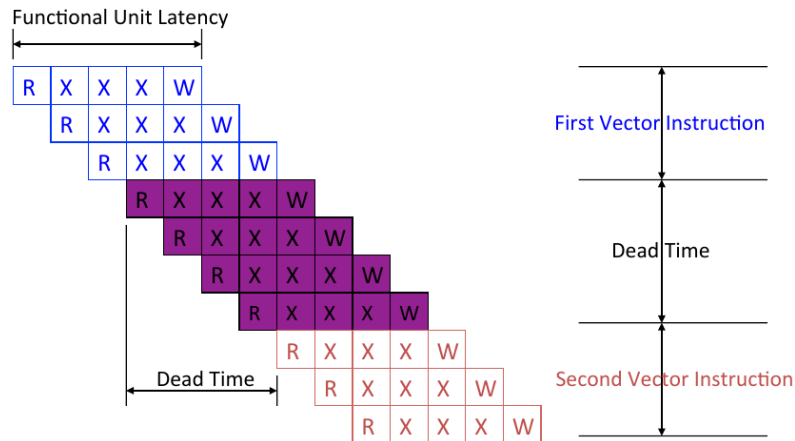


- Pipeline stalls decreases: stalls only for first vector element
- Convoy: set of vector instr that could potentially execute together (no structural hazards).

For a vector of length n and m convoys then $n*m$ clock cycles are required.

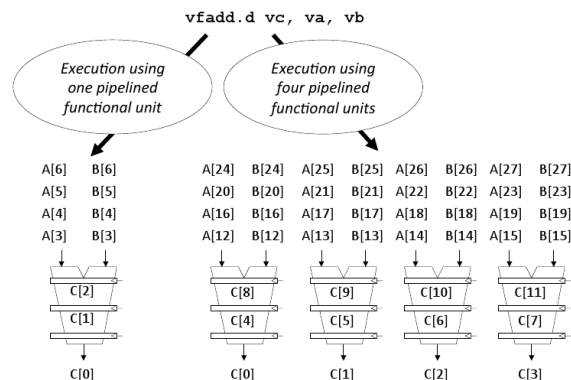
Vector startup components:

- FU latency (time through pipeline)
- dead time or recovery time (time before another vector instruction can start down pipeline)



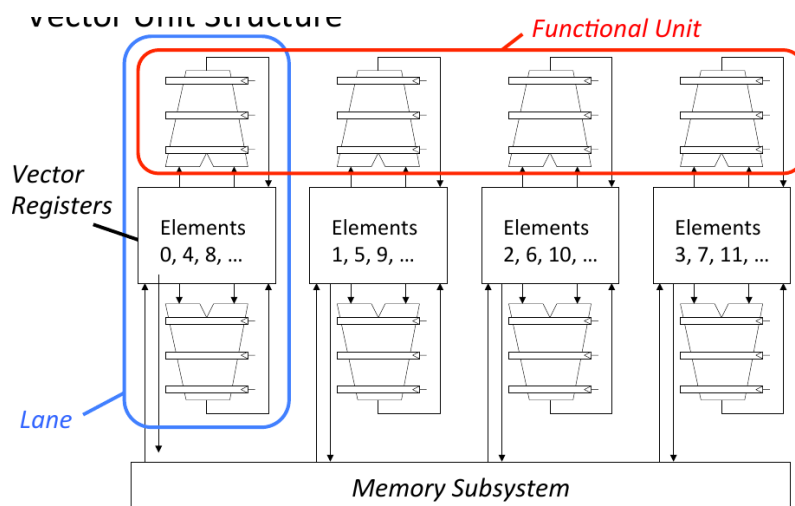
Improvements:

- **Vector instr execution with multiple lanes:** more than one FU that works in parallel on same operation.



- **Interleaved vector memory system:** let the FU to be loaded with interleaved vector data.

Access to data with base+stride.



- **Automatic code vectorization:** loops can be unrolled and paired operations together in

order to parallelize operations on loops. Stripmining: since registers have finite length, it is necessary to break them into smaller pieces that fit into the registers.

- **Vector Conditional Execution:** since loops can have conditioned parts, then it is useful to put flags (“vector mask”) that turn operations into nops if they don’t have to be executed.

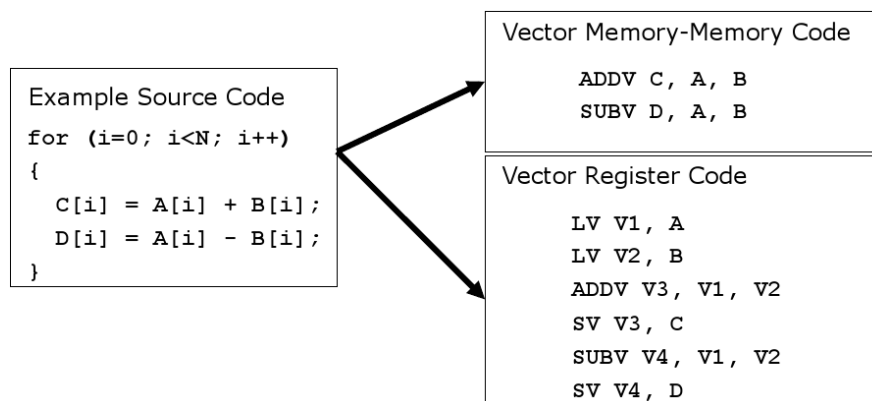
Masked vector instr can be:

- Simple implementation: execute all N operations, turn off result WB according to mask
- Density-time implementation: execute only elements non-zero masks

- **Vector Memory-Memory (VMMA) vs Vector Register Machines**

VMMA arch require greater main memory bandwidth because they must be read from memory. Overlap execution more difficult since must check dependencies on memory addresses. Also greater startup latency.

Vector provide efficient execution of data-parallel loop codes. Scales to more lanes without changing the binary code. Provide fast temporary storage to reduce memory bandwidth demands and simplify dependence checking. But requires extensive compiler analysis to be certain that loops can be vectorized.



- **Multimedia Extensions (or SIMD extensions):** use existing registers with sub-partitioning. Single instruction operates on all the data contained in a single register.
 - limited instr set
 - limited vector register length

MIMD

Each processor fetches its own instr and operates on its own data.

Processors are often off-the-shelf microprocessors, can scale to a variable number of cpu nodes.

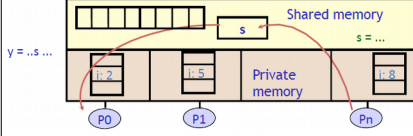
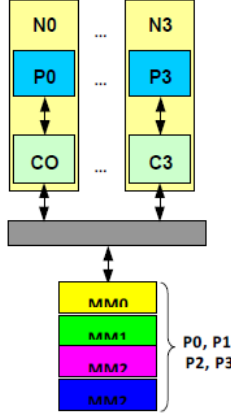
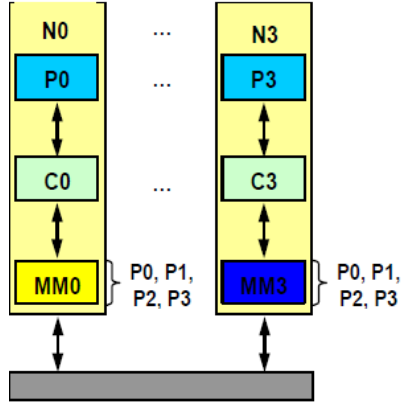
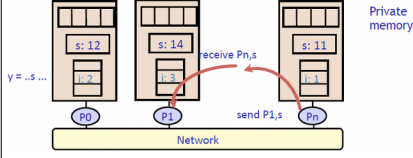
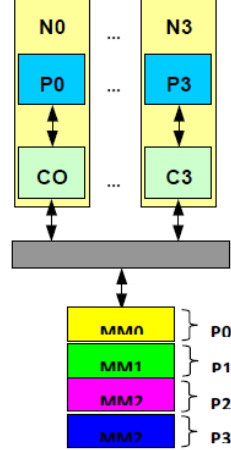
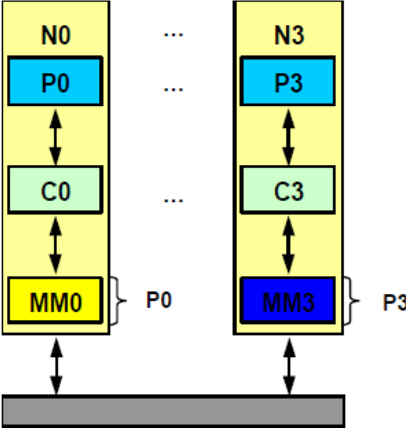
Flexible:

- single-user machines with focus on high-performance for one specific applications
- multi-programmed machines running many tasks simultaneously

problems:

- fault tolerancy
- to exploit MIMD with n processors, at least n threads or processes to execute

MIMD machines types:

	Centralized memory <ul style="list-style-type: none"> • supports few processors • large caches • Also called SMP (Symmetric MultiProcessors) with UMA (Uniform Memory Access) architecture style. 	Distributed memory <ul style="list-style-type: none"> • support many processors • large bandwidth interconnection. • Complex data communication between processors. • Non Uniform Access Memory (NUMA).
<p><i>Single logically shared address space (shared memory architecture)</i></p> <p>processors communicate with shared address space.</p> 		
<p><i>Multiple and private address spaces (message passing architectures)</i></p> <p>processors have private memories, communicate via messages. More complex to program but explicit communication.</p> 		

Shared Memory Architectures

Advantages:

- Easy programming (implicit communication)
- low latency
- easy HW controlled cache

Disadvantages:

- complex to build scalable systems
- requires sync ops
- hard to control data placement in cache

Most used architecture. Uniform access via load/store. Normal uniprocessor way to access data (through memory hierarchy).

Caching

2 main types of machines:

- non cache coherent
- HW cache coherent

cache increases bandwidth and reduces latency access.

Caching of both, private (used by single processor) and shared data (used by multiple processors).

Cache coherency problem

Shared cache may be replicated among multiple processors → problem of cache coherency
coherence through:

- access shared data always through memory → slow but easy
- read in parallel but processor lock on write. Coherence protocol need to check all caches to update values → fast but complex

Consistency and coherence are complementary.

Coherence

Any write must eventually be seen by a read. Write always seen in proper order (serialization)

2 rules to enforce coherence:

- “if P writes x & P1 reads x → P write will be seen by P1 if read and write are sufficiently far apart”
- write to a single location are serialized: 2 writes to same location by any 2 processors are seen in the same order by all processors → latest write will be seen (otherwise illogical order)

→ Defines behavior of reads and writes to the same memory location. Ensures that writes by one processor are eventually visible to other processors, for one memory address.

Consistency

System is sequentially consistent if: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program (no instruction reordering).

→ Defines behavior of reads and writes wrt accesses of other memory locations. States when a write by one processor can be observed by a read on another, across different addresses.

Coherent multiprocessor

what it offers:

- *Migration*: data item can be moved to a local cache and used there in a transparent fashion (reduce access latency)
- *Replication*: for shared data that are being simultaneously read (caches make a copy of the data item in the local cache)

how it ensures coherence (“cache coherence protocols”):

- *Snooping*: all cache controllers keep track (snoop) the status of owned block of memory. In SMP the caches are accessible via some broadcast medium (bus), and all cache controllers check the medium for request for data they have, in which case they respond accordingly to

ensure coherence (invalidate, update, supply new value).

Snooping may cause interference to processor cache access → processor stalls. To reduce this an extra port is added to the address tag portion of the cache

snoopy cache coherence protocols:

- Write invalidate protocol: processor has exclusive access to data item before it writes that item: invalidate address in all other caches (by broadcasting a signal on the bus) before performing write. All caches on the bus check and invalidate the relative block. Only done for the first write (similar to WB) thus reducing the traffic.
 - Write-through: memory always up to date, so data is taken always from memory. Lot of bandwidth used, but every write is observable
 - Write-back: other processor's private cache may have updated data → snoop in other caches

Multiple readers but only one writer.

- MSI protocol (write-invalidate, write-back):

	States		
Memory block	<i>Shared</i> in all caches and up-to-date in memory	<i>Modified</i> in exactly one cache	<i>Uncached</i> not in cache
Cache block	<i>Shared</i> block not modified and can be read	<i>Modified</i> cache has only one writeable/dirty copy	<i>Invalid</i> no valid data

- MESI protocol (enhanced wrt MSI): add exclusive state

	States	
Cache block	<i>Exclusive</i> copy only available in one cache (not necessary to send invalid signal through the bus, known that owner has exclusive access)	Same as MSI

- Write update protocol: writing processor broadcasts the new data over the bus, all caches check if they have a copy of the data and if so they update it. Similar to write through protocol since all the caches are updated. Reduced latency because data is already available.

Snooping with level2 cache: inclusion property: entries in L1 must be in L2 (invalidation in L2 → invalidation in L1)

False sharing: cache coherence is done at line level and not word level. Line can be invalidated many times because it suffices only 1 modification.

state	line addr	data0	data1	...	dataN
-------	-----------	-------	-------	-----	-------

Performances: scaling up means more communication bandwidth over bus and snoop bandwidth into tags. Multiple interleaved buses decongestion the bus.

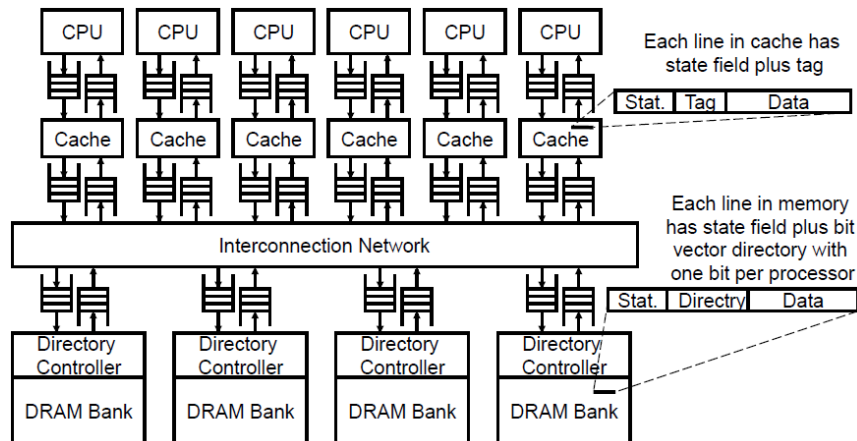
Buses don't scale (point-to-point network help to scale) and most snoops fail to find a match. Absence of a centralized data structure that tracks the state.

- *Directory-Based*: directory keeps track of status of a block in one location, scales better.

Message oriented protocol: requests generate messages sent between nodes and receive explicit responses.

Each memory line has associated directory information.

On a miss communicate with directory entry and communicate only with the nodes that have copies.

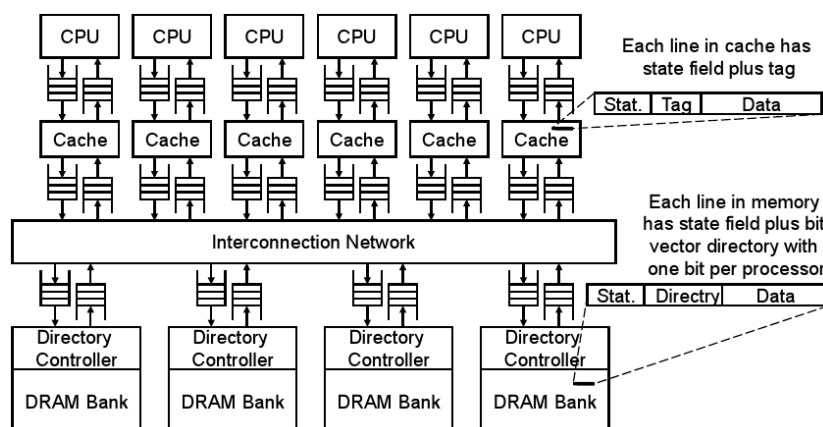


Directory maintains info regarding:

- state: state of each block
 - uncached: no cpu has a copy in cache/block not valid in any cache
 - shared: 1/more cpu have cache block and memory is up to date. Sharer set of proc ID
 - modified: only one cpu (owner) has modified data (memory is out-of-date)
- copy: which processor has a copy (flag)
- owner: processor owner of the block (flag)

If shared L3 cache: easy to implement a directory scheme

keep bit vector of size equal to the number of cores for each L3 block (set invalidation only on caches)



2 primary operations:

- handle read miss →
- handle write to shared clean block →
- handle write miss → read miss + write to shared block

Directory protocol messages:

Type	Source	Destination	Content
Read miss	Local cache	Home dir	P, ADD
	P read miss at addr ADD; request data and makes P a read sharer		
Write miss	Local cache	Home dir	P, ADD
	P has a write miss at addr ADD; request data and make P the exclusive owner		
Invalidate	Local cache	Home dir	ADD
	Request to send Invalidates to all remote caches that are caching the block at address ADD		
Invalidate	Home dir	Remote cache	ADD
	Invalidate a shared copy of data at address ADD in all remote caches		
Fetch	Home dir	Remote cache (owner)	ADD
Fetch/Inv.	Fetch the block in home at address ADD and owner send data to its home dir (through Data WB); change the state of ADD in the Remote cache from Modified to Shared. (Block state in home changes to shared)		
	Home dir	Remote cache (owner)	ADD
Data value reply	Fetch the block in home at address ADD and owner send data to its home directory (through WB); Invalidate the block in the remote cache. (Block state in home stays Modified – owner changed)		
	Home dir	Local cache	Data
	Return a data value from the home memory back to the requesting node (Miss response)		
Data WB	Remote cache (owner)	Home dir	ADD, Data
	WB a data value for address ADD from remote cache owner to the home directory (fetch and fetch/invalidate response)		

Cache e memory states in the home directory:

	States			
Memory line	<i>R(dir)</i> memory line is shared by the sites specified in dir. The data in memory is valid in this state. If memory empty then line is not cached by any site.	<i>W(id)</i> memory line is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.	<i>TR(dir)</i> memory line is in transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.	<i>TW(id)</i> memory line is in transient state waiting for a line exclusively cached at site id to make the memory line at the home site up-to-date
Cache line	<i>C-invalid</i>	<i>C-shared</i>	<i>C-modified</i>	<i>C-transient</i>

	in all caches and up-to-date in memory	the accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid	(=Execution) the accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data	the accessed data is in a transient state
--	--	--	---	---

Message Passing Architectures

Advantages:

- Easy designated (explicit communication)
- easy to control data placement in cache

Disadvantages:

- message passing overhead high
- complex to program
- reception techniques problems (interrupts/polling)

While sending messages is cheap, receiving is costly (interrupt/polling).

Concurrency Management

Protocol would be easy to design if only one transaction in flight across entire system.

Great complexity in managing multiple outstanding concurrent transactions to cache lines.

Memory Consistency Problem

In what order must a processor observe the data writes of another processor?

Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processor were interleaved → simple solution: delay. → implementation: wait for stores to complete before issuing other stores

relaxed memory models: SC requires to maintain all possible orderings $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$. The relaxed models are defined by which of these 4 sets of orderings they relax (not all are supported though, depends on the memory model).

Synchronization

sync arises whenever there are concurrent processes in a system, 2 types:

- **Producer-Consumer:** Consumer process waits until the producer process has produced data
- **Mutual exclusion:** ensures only one process uses a resource at a give time

Release Consistency

Consistency matters only when processes communicate data.

