$$\text{Execution Time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}}$$

Ins per program:program,compiler,ISA.Seconds per cycle:micro-arch,tech. CPI:program,compiler,ISA,micro_arch. Include in ISA: instruction set; regs, mem; operating modes. Not in ISA: Op implement, op speed, op power, mem implement, ? cache.
Good ISA: programmability, implementability, compatibility.

Certain ISA features make these difficult
- Variable instruction lengths/formats: complicate decoding
- Implicit state: complicates dynamic scheduling
- Variable latencies: complicates scheduling
- Difficult to interrupt instructions: complicate many things
  - Example: memory copy instruction

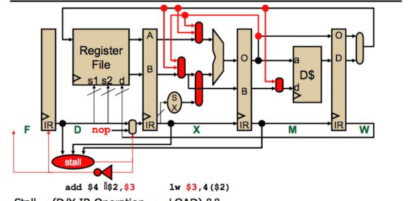
	insns/program	cycles/insn	seconds/cycle	other
CISC	↓	↑	↑	+ Easy for assembly-level programmers + good code density
RISC	↑ hopefully not *too* much	↓	↓ *if designed aggressively*	+ smart compilers can help with insns/program

Latency: time to finish a fixed task. Throughput: # tasks in fixed time.
CPI = CPU time / (clock period * dynamic isn count)
Amdahl's Law: Make the common case fast.
Reduce Dynamic Power: # transistors ↓, capacitance ↓, volt ↓, freq ↓, activity ↓.
Reduce Static Power: # transistors ↓, volt ↓, disable transistors, dual volt, low-leakage. Moore's Effect on Power: reduce power/transistor, increases power density and total power. Use low-leakage transistors reduce both dynamic and static power.

What metric would you use to compare the performance of computers

1. With different ISAs?	Execution time
2. With the same ISA?	MIPS
3. With the same ISA and clock speed?	IPC

Stalling on Load-To-Use Dependences



`add $4 ←$2,$3` `lw $3,4($2)`

Stall = (D/X.IR.Operation == LOAD) &&
((F/D.IR.RegSrc1 == D/X.IR.RegDest) ||
((F/D.IR.RegSrc2 == D/X.IR.RegDest) && (F/D.IR.OP != STORE))

Dependence: data&control. load&use dependency, add stall.
Variable ins length and format make F&D difficult. Implicit state makes dynamic scheduling difficult. Variable latencies makes scheduling difficult.
Control Hazard: 2 cycle penalty.

	1	2	3	4	5	6	7	8	9
`mul $4 ←$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6 ←$4,1`		F	d*	d*	d*	D	X	M	W

	1	2	3	4	5	6	7	8	9
`mul $4 ←$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4 ←$1,1`		F	d*	d*	D	X	M	W	
…									
…									
`add $10 ←$4,$6`					F	D	X	M	W

State/prediction	BHR=NNN	N*	T	T	T	T	T	T	T	T	T	T	T
	BHR=NNT	N	N*	T	T	T	T	T	T	T	T	T	T
	BHR=NTN	N	N	N	N	N	N	N	N	N	N	N	N
"active pattern"	BHR=NTT	N	N	N*	T	T	T	T	T	T	T	T	T
	BHR=TNN	N	N	N	N	N	N	N	N	N	N	N	N
	BHR=TNT	N	N	N	N	N	N*	T	T	T	T	T	T
	BHR=TTN	N	N	N	N	N*	T	T	T	T	T	T	T
	BHR=TTT	N	N	N	N	N	N	N	N	N	N	N	N
Outcome	N N N	T	T	T	N	T	T	T	N	T	T	T	N

- Last unit: **pipeline-level parallelism**
  - Execute one instruction in parallel with decode of next
- Next: **instruction-level parallelism (ILP)**
  - Execute multiple independent instructions fully in parallel
  - Today: multiple issue
  - In a few weeks: dynamic scheduling
    - Extract much more ILP via out-of-order processing
- **Data-level parallelism (DLP)**
  - Single-instruction, multiple data
  - Ex: one instruction, four 16-bit adds (using 64-bit registers)
- **Thread-level parallelism (TLP)**
  - Multiple software threads running on multiple cores

Global BHR captures local pattern for tight loop branches.

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r4,r5→r6`		F	d*	d*	D	X	M	W				
`add r2,r3→r7`			F	p*	D	X	M	W				
`add r7,r6→r8`					F	D	X	M	W			
`lw 0(r8)→r9`					F	d*	D	X	M	W		

Avoid N^2 Bypass: Clustering, full bypassing within cluster.
Wide Non-Sequential Fetch, compiler can help.
VLIW: + Simpler i$/branch prediction, + Simpler dependence check logic, - Not compatible across machines of different widths.
Static scheduling by the compiler, dynamic scheduling by the hardware.
Utilization: actual performance / peak performance

SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ldf X(r1)→f1`	F	D	X	M	W															
`mulf f0,f1→f2`	F	D	d*	d*	E*	E*	E*	E*	E*	W										
`ldf Y(r1)→f3`		F	D	X	M	W														
`addf f2,f3→f4`		F	p*	p*	d*	d*	d*	d*	D	E+	E+	W								
`stf f4→Z(r1)`			F	p*	p*	p*	p*	p*	p*	D	X	M	W							
`addi r1,4→r1`					F	p*	p*	p*	p*	p*	D	X	M	W						
`blt r1,r2,0`					F	p*	p*	p*	p*	p*	d*	D	X	M	W					
`ldf X(r1)→f1`											F	D	X	M	W					

2-way superscalar pipeline
- Any two insns per cycle + split integer and FP pipelines
- + **Performance**: 7 insns / 10 cycles = 0.70 IPC
- − **Utilization**: actual/peak IPC = 0.70 / 2 = 35%
- − More hazards → more stalls
- − Each stall is more expensive

Unrolled SAXPY Performance/Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ldf X(r1)→f1`	F	D	X	M	W															
`ldf X+4(r1)→f5`		F	D	X	M	W														
`mulf f0,f1→f2`			F	D	E*	E*	E*	E*	E*	W										
`mulf f0,f5→f6`				F	D	E*	E*	E*	E*	E*	W									
`ldf Y(r1)→f3`					F	D	X	M	W											
`ldf Y+4(r1)→f7`						F	D	X	M	s*	s*	W								
`addf f2,f3→f4`							F	D	d*	E+	E+	s*	W							
`addf f6,f7→f8`								F	p*	D	E+	p*	E+	W						
`stf f4→Z(r1)`										F	D	X	M	W						
`stf f8→Z+4(r1)`											F	D	X	M	W					
`addi r1→8,r1`												F	D	X	M	W				
`blt r1,r2,0`													F	D	X	M	W			
`ldf X(r1)→f1`														F	D	X	M	W		

- + Performance: 12 insn / 13 cycles = 0.92 IPC
- + Utilization: actual/peak IPC = 0.92 /1 = 92%
- + **Speedup**: (2 * 11 cycles) / 13 cycles = 1.69

For Loop: loop unrolling; No-For Loop: superblock(biased block), predication(no biased)
Loop unrolling: schedule 2+ iterations together, Fuse iterations, Schedule to reduce stalls, Schedule introduces ordering problems, rename registers.

– Static code growth more I$ misses (limits unrolling)

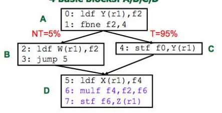
– Needs more registers to hold values (ISA limits this)
– Doesn't handle: non-loops, inter-iteration dependences

Superblocks

Source code
```
A = Y[i];
if (A == 0)
   A = W[i];
else
   Y[i] = 0;
Z[i] = A*X[i];
```

Machine code
```
0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

4 basic blocks: A,B,C,D



- Use when branch is highly biased
- Fuse blocks of most frequent path: ACD
- Schedule
- Create **repair code** in case path = ABD

If-conversion: replacing control with predication
+ Good if branch is unpredictable (save mis-prediction)
– But more instructions fetched and "executed"
Benefit:predication avoids branches Thus avoiding mis-predictions Also reduces pressure on predictor table (few branches to track)

Cost:extra(annulled)instructions

ISA Support for Predication

```
0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
```

- IA-64: change branch 1 to **set-predicate insn `fspne`**
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `p1` is true
  - `stf.np` performs `stf` if predicate `p1` is false

Scheduling: Compiler or Hardware

Compiler
- + Large scheduling scope (full program)
- + Simple hardware → fast clock, short pipeline, and low power
- − Low branch prediction accuracy (profiling?)
- − Little information on memory dependences (profiling?)
- − Can't dynamically respond to cache misses (or anything really)
- − Hard to speculate, recover from mis-speculation (h/w support?)

Hardware
- − Finite buffering resources fundamentally limit scheduling scope
- − Scheduling machinery adds pipeline stages and consumes power
- + High branch prediction accuracy
- + Dynamic information about memory dependences
- + Can respond to cache misses
- + Easy to speculate and recover from mis-speculation

Tag overhead of 32KB cache with 1024 x 32B entries
- 32B blocks → 5-bit offset
- 1024 entries → 10-bit index
- 32-bit address → 32-bits – (5-bit offset + 10-bit index) = 17-bit tag
  (17-bit tag + 1-bit valid) X 1024 entries = 18Kb tags = 2.2KB tags
  ~6% overhead

Tag|index|offset

Classifying Misses: 3C Model (Hill)

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of entries in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

Victim Buffer: On miss, check VB; hit? Place block back in I$/D$. Shared among all sets.%miss no change, increase latencymiss.
Lockup free: allows other accesses while miss is pending.
Software Restructuring: Capacity misses.
Prefetching: put blocks in cache proactively/speculatively.

Option #1: Write-through: immediately
• On hit, update cache
• Immediately send the write to the next level
• Option #2: Write-back: when block is replaced
• Requires additional "dirty" bit per block
• Replace clean block: no extra traffic
• Replace dirty block: extra "writeback" of block
+ Writeback-buffer (WBB): keep it off critical path
1. Send "fill" request to next-level
2. While waiting, write dirty block to buffer
3. When new blocks arrives, put it into cache
4. Write buffer contents to next-level

Write miss:address not in cache.
Write-no-allocate: the write op goes directly to MM without affecting the cache. Good idea if the data not immed used.
Write-allocate: update MM and cache. Good idea if data needed again soon.

Write Propagation Comparison

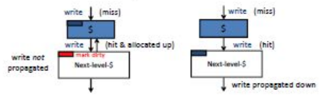
- **Write-through**
  - − Requires additional bus bandwidth
    - Consider repeated write hits
  - − Next level must handle small writes (1, 2, 4, 8-bytes)
  - + No need for dirty bits in cache
  - + No need to handle "writeback" operations
    - Simplifies miss handling (no write-back buffer)
  - Sometimes used for L1 caches (for example, by IBM)
- **Write-back**
  - + Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used by Intel and AMD
  - 2nd-level and beyond are generally write-back caches

Write Miss Handling

How is a write miss handled?

Write-allocate: fill block from next level, then write it
- + Decreases read misses (next read to block will hit)
- − Requires additional bandwidth
- Commonly used (especially with **write-back** caches)

Write-non-allocate: just write to next level, no allocate
- − Potentially more read misses
- + Uses less bandwidth
- Use with **write-through**



- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM} = 1ns+(0.05*50ns) = 3.5ns$
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2} + \%_{missL2} * t_{hitM} = 10ns+(0.2*50ns) = 20ns$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{avgL2} = 1ns+(0.05*20ns) = 2ns$

Non-shared: check its own history pattern. Shared: Merge the history pattern.

```
loop:  load   r1, 0(r2)        (1)
       addi   r1 ← r1, #1      (2)
       store  0(r2), r1        (3)
       addi   r2 ← r2, #4      (4)
       sub    r4 ← r3, r2      (5)
       bnez   r4, loop         (6)
       next instruction        (7)
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(1)	F	D	X	M	W													
(2)		F	D	d*	d*	X	M	W										
(3)			F	p*	p*	D	d*	d*	X	M	W							
(4)						F	p*	p*	D	X	M	W						
(5)									F	D	d*	d*	X	M	W			
(6)										F	p*	p*	D	d*	d*	X	M	W
(7)													F	p*	p*	D	f*	
(1)																	F	D

In the table below, columns labeled P show the value of a 1-bit predictor shared by B1 and B2. Columns labeled B1 and B2 show the actions of the branches (each alternating taken/not taken). Time increases to the right. T stands for taken, NT for not taken. The predictor is initialized to NT.

	P	B1	P	B2	P	B1	P	B2	P	B1	P	B2	P	B1	P	B2
	NT	T	T	NT	NT	NT	NT	T	T	T	T	NT	NT	NT	NT	T
Correct?		no		no		yes		no		yes		no		yes		No

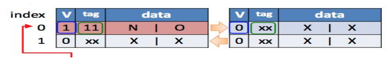
Because a single predictor is shared, prediction accuracy improves from 0% to 50%.

Here, B1 is always taken, B2 is always not taken, and they are interleaved as in (a).

	P	B1	P	B2	P	B1	P	B2	P	B1	P	B2	P	B1	P	B2
	NT	T	T	NT	NT	T	T	NT	NT	T	T	NT	NT	T	T	NT
Correct?		no		no		no		no		no		no		No		No

If each had a 1-bit predictor, each would be correctly predicted after the initial startup transient. Because a single predictor is shared, accuracy is 0%.

Arithmetic Mean: For units that are proportional to time (e.g., latency).
Harmonic Mean: For units that are inversely proportional to time (e.g., throughput).
Geometric Mean: For unitless quantities (e.g., speedup ratios).



+ Associativity:
↓conflict misse
↑hit time ☹

+ Capacity:
↓capacity misses ☺
↑hit time ☹

+ Block Size:
↓cold misses ☺
↑conflict misses ☹

Read-after-write (RAW),Write-after-read (WAR),Write-after-write (WAW)
Register Renaming Algorithm: Rename the output, Once all older instructions have committed, free register.

Original insns

```
xor r1, r2 → r3
add r3, r4 → r4
sub r5, r2 → r3
addi r3, 1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
```

Map table

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Free-list

p8
p9
p10

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Age
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	n	---	y	p9	3

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	**n**

• ROB entry holds all info for recover/commit • Logical register names
• Physical register names• Instruction types

Original insns

```
bnz r1 loop
xor r1, r2  → r3
add r3, r4 → r4
sub r5, r2 → r3
```

Renamed insns

```
bnz p1, loop
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
```

Overwritten Reg

```
[  ]
[p3]
[p4]
[p6]
```

Map table

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Free-list

p8
p9
p10

Original insns

```
xor r1,r2 → r3
add r3,r4 → r4
sub r5,r2 → r3
addi r3,1 → r1
```

Renamed insns

```
xor  p1, p2 → p6
add  p6, p4 → p7
sub  p5, p2 → p8
addi p8, 1  → p9
```

Overwritten Reg

```
[p3]
[p4]
[p6]
[p1]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

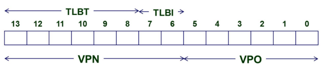
p10
p3

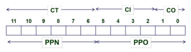
Instruction	Disp	Issue	WB	Commit
`ld [p1] → p2`	1	2	5	6
`add p2, p3 → p4`	1	5	6	7
`xor p4, p5 → p6`	2	6	7	8
`ld [p7] → p8`	2	3	6	8

Cycle 8:
• xor and ld can commit (2-wide: can do both at once)

Dynamically Scheduling Memory Ops,Options for hardware:
• Holdloadsuntilallpriorstoresexecute(conservative)
• Execute loads as soon as possible, detect violations (aggressive)
• When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
• Learn violations over time, selectively reorder (predictive)
Store→Load Forwarding: • Get value from executed (but not comitted) store to load Load Scheduling: • Determine when load can execute with regard to older stores
Store Queue: handles forwarding. Load Queue: detects ordering violations. Both together • Allows aggressive load scheduling
Window Size: Constrained by physical registers(#preg). Constrained by issue queue. Constrained by load+store queues.
CGMT: + Sacrifices little single thread performance (of 1 thread) – Tolerates only long latencies (e.g., L2 misses)
FGMT: – Sacrifices significant single thread performance + Tolerates latencies (e.g., L2 misses, mispredicted

- 32-bit machine → 32-bit VA → 4GB virtual memory (2^{32}=4G)
- 4GB virtual memory / 4KB page size → 1M VPs
- Each VP needs a PTE: 1M VPs → 1M PTEs
- 1M PTEs * 4B-per-PTE → **4MB**





Page fault: PTE not in TLB or page table

```
mov r1->r2
ld r1,0(&lock)
st r2,0(&lock)
```

```
acquire sequence
(value of r1 is 1)
 A0: swap r1,0(&lock)
 A1: bnez r1,A0
```

Solution: **test-and-test-and-set locks**
- New acquire sequence

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: swap r1,0(&lock)
A4: bnez r1,A0
```

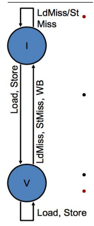
Processors can spin on a busy lock locally (in their own cache) + Less unnecessary interconnect traffic.
Queue lock: Each waiting processor spins on a different location (a queue)
+ Greatly reduced network traffic (no mad rush for the lock)
+ Fairness (lock acquired in FIFO order)
– Higher overhead in case of no contention (more instructions)
– Poor performance if one thread gets swapped out
Coarse-grain locks: correct, but slow. Fine-grain locks: parallel, but difficult.

Processor 0

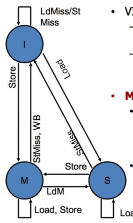
```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```

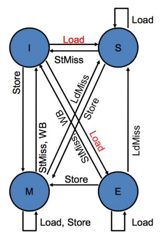


CPU0	CPU1	Mem
		500
V:500		500
V:400		500
I:	V:400	400
	V:300	400



- VI protocol i
  - Only one system
  - Multiple
    - Not a p
    - Big pro
- **MSI (modi**
  - Fixes pro states
    - **M (mo**
    - **S (sha**
  - Allows **ei**
    - Multipl
    - Single

CPU0	CPU1	Mem
		500
S:500		500
M:400		500
S:400	S:400	400
I:	M:300	400



CPU0	CPU1	Mem
		500
E:500		500
M:400		500
S:400	S:400	400
I:	M:300	400

MSI Directory Protocol

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```

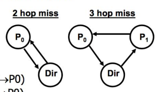
P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
I:	M:300	M:1:400

`ld` by P1 sends BR to directory
- Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**

`st` by P1 sends BW to directory
- Directory sends BW to P0, P0 goes to **I**

Directory Flip Side: Latency

- Directory protocols
  + Lower bandwidth consumption → more scalable
  – Longer latencies
- Two read miss situations
- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, many processors → high probability someone has it



Directories: non-broadcast coherence protocol
Processor sends coherence event to home directory
• Home directory only sends events to processors that care
- Bus-based snooping: all processors see all requests in same order
  - Ordering automatic
- Point-to-point network: requests may arrive in different orders
  - Directory has to enforce ordering explicitly
  - Cannot initiate actions on request B…
    ..until all relevant processors complete actions on request A
  - Requires directory to collect acks, queue requests, *etc.*