CPI Info

Effective CPI = sum(CPI ideal * instruction count) / overall IC
CPU time = IC X CPI ideal * clock cycle (or / clock rate)
CPU time with memory = IC * CPI stall * CC
CPI stall = CPI ideal + access rate * miss rate * miss penalty
AMAT = time for hit + MR * MP (measured in time, not cycles)

AMAT w/ mutli cache:

Lx MP = Lx + 1 hit time + Lx + 1 MR * lx + 1 MP,

Locality

Temporal: recently accessed likely to be accessed soon
Spacial: adjacent items likely to be used in near future
C = row major, stride n access = access x + n data
Inlined has good locality, inserted sequentially into instructions

Cache Policies / Misses

Mapping policies decide where new blocks go (usually b mod n, overwrite whatever is in slot if all are full)
Replacement policy also dictates what is removed usually LRU or FIFO)
Cold / compulsory miss occurs at first data access (always)
Capacity miss occurs when active blocks (actively being used by program) is larger than size of cache at level k
Conflict miss occurs when multiple data from cache k + 1 map to the same slot in cache k (i.e. 0 and 8 map to same place in mod 4 mapping, causes thrashing)

Advanced Spatial Locality Example

Calculate the number of cache hits and cache misses given: a has row-major INT4 elements and is cache-aligned.
Cache capacity of 4 Kilobytes, cache block is 32 bytes. All local variables stored in registers, not cache.
int i, sum = 0, N = 16384;
for (i = 0; i < N; i++) sum += a[i];
Stride-1 access to a. Since cache block is 32 bytes, each block holds 32/4 = 8 ints. One cold miss for each block.
(1/8) × 16384 = 2048 cache misses. 16384- 2048 cache misses = 14336 cache hits.

Memory Cache Mapping

Cache is array of R = 2^s sets, each set containing N>=1 lines, each line holding B = 2^b bytes of data
Capacity = B * N * R
When memory words are more than 1 byte, we're left with different byte memory addresses and wor memory addresses for blocks of data -> word address can be changed to byte address with N * X, where N = bytes per word and X = word address

Address Mapping

To determine tag, set index and block offset for accessing an N-way cache, must know cache organization
Direct-Mapped
N =1 -> One line per set, Each memory address is mapped to exactly one line in the cache.
b = log2B, R=C/B, s=log2R, t (tag size) =m−s−b.
Fully Associative
R =1(allow a memory address to be mapped to any cache block). Tag is whole address except block offset. b = log2B, N =C/B, s=0, t=m−b.
N-way set associative
N is typically 2, 4, 8, or 16. A memory block maps to a specific set but can and can be placed in any way of that set (so there are N choices of mapping).
b = log2B, R=C/B×N , s=log2R, t=m−s−b.

Handling Cache Hits and Misses

Read Hits ⇒ Do nothing special.
Write Hits (Data only) has two policies:
Write-through: Require cache and backing memory to always be consistent.
-> When writing to a cache, also pass the value to the next lower level cache (or main memory) to update its copy.
->In naïve implementation this is very slow. Speed of cache writes limited by the speed of the next lower level.
-> Use write buffer between levels, stall only if buffer is full.
Write-back: Allow cache and memory to be inconsistent
-> Write data into cache block, this becomes a dirty block.
-> Only update next lower level when a dirty block is evicted.
-> Requires >two cycles– one to check for evict/dirty and another to actually do write– or again use a write buffer and use only one cycle.
Read Misses (Instruction and Data)
Stall the execution, fetch block from the next lower level of memory, write ("install") into cache, pass to next higher level of memory (or the processor), and let processor resume.
Write Misses (Data only) stall execution, perform one of two
Write allocate: Fetch block from next level in hierarchy
-> Install it in cache, write updated word into the new block
-> Installing + writing updated word can be done same time.
No-Write Allocate: Skip fetching/installing block into cache, write directly into lower level of memory (or a write buffer). then let the processor resume.
Compulsory Misses: Caused by cold starts, process migrations or very first references. Reduce impact by increasing block size. But this causes increased miss penalty and could increase miss rate.
Capacity Misses: Caused by the cache becoming full; it cannot hold all blocks referenced by the program. Reduce impact by increasing cache size (may increase access time)
Conflict Misses: Caused by multiple addresses mapping to the same cache block. Reduce impact by increasing associativity or increasing cache/block size (but may increase access time) ë Larger cache ⇒ more sets ⇒ fewer addresses map to same loc.

Cache Performance Improvements

Reducing Cache Miss Rate ⇒ increase cache size
With increasing technology (especially transistor size and density) there is more room for larger caches.
Reducing Cache Miss Penalty ⇒ use more levels of cache L1 cache around for a long time.
Recall: New AMAT Example 1 cycle L1 hit time, 2% L1 miss rate, 5 cycle L2 hit time, 5% L2 miss rate,100 cycle main memory access time
Without L2 cache: AMAT = 1 + .02*100 = 3
With L2 cache: AMAT = 1 + .02*(5 + .05*100) = 1.2
Multilevel Cache Design
Design considerations for L1 and L2 caches are very different: Primary cache should focus on minimizing hit time in support of faster processor clock.
-> Smaller capacity with smaller block sizes.
Secondary cache(s) should focus on reducing miss penalty for L1 cache by reducing miss rate to main.
-> Larger capacity with larger block sizes.
-> Higher levels of associativity.
The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache.
-> L1 cache can be smaller and faster (but results in higher miss rate). Hit time is less important for L2 cache than miss rate.
- >But, L2 hit time determines L1's miss penalty.
-> Presence of L3 cache greatly improves the situation, allowing L2 miss rate to be slightly less important.
Improving Cache Performance (1/2)
(1) Reduce the time to hit in the cache.
-> Smaller cache size, smaller block size.
-> Direct-Mapped
-> For writes, two possible strategies:-
No-Write Allocate: no "hit" on cache, just write to write buffer. Makes subsequent reads tricky.-
Write Allocate: avoid 2 cycles using write buffer to write to lower cache.
(2) Reduce the miss rate.
-> Larger cache, Larger block size
-> More flexible placement (increase associativity).
-> Use a victim-cache– a small buffer holding most recently discarded blocks.
Improving Cache Performance (2/2)
(3) Reduce the miss penalty
-> Smaller blocks.
-> Use a write-buffer.
-> Check the write-buffer (or the victim-cache) on a read miss: luck!
-> Pre-fetch critical word first, then rest of cache block.
-> Use multiple cache levels.
->Faster backing store (= main memory).
-> Improved memory bandwidth– amount and speed of memory transfer between levels (e.g. wider buses).

Boolean Algebra Laws

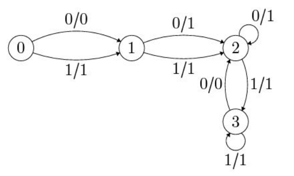
Associativity: $(A+B)+C \equiv A+(B+C)$, $(A \cdot B) \cdot C \equiv A \cdot (B \cdot C)$
Commutativity: $A+B \equiv B+A$, $A \cdot B \equiv B \cdot A$
Distributivity: $(A+B) \cdot C \equiv (A+B) \cdot (A+C)$, $A \cdot (B+C)$
$\equiv A \cdot B+A \cdot C$
Identity: $A+0 \equiv A$ $A \cdot 1 \equiv A$
Annihilation: $A+1 \equiv 1$ $A \cdot 0 \equiv 0$
Idempotence: $A+A \equiv A$ $A \cdot A \equiv A$
Absorption: $A \cdot A+B \equiv A$, $A+A \cdot B \equiv A$
Double Negation $\sim\sim A \equiv A$
Complementation: $A+\sim A \equiv 1$ $A \cdot \sim A \equiv 0$ l
De Morgan's Laws: $\sim(A+B) \equiv \sim A \cdot \sim B$, $\sim(A \cdot B) \equiv \sim A+\sim B$
Canonical Forms
Conjunctive Normal Form (CNF) ⇒ AND of ORs
Disjunctive Normal Form (DNF) ⇒ ORs of ANDs
CNF= $a+b \cdot \sim a+b \cdot \sim a+\sim b$, DNF = $ab+\sim ab+\sim a\sim b$
Every variable should appear in every sub-expression.
Functional Completeness
A set of functions (operators) which can adequately describe every operation and outcome in an algebra.
For Boolean algebra the classical set of operators: +, ·, ¬ is functionally complete but not minimal. Thanks to De Morgan's Law only need one of AND or OR. The sets +, ¬ and ·,¬ are both functionally complete and minimal.
-> minimal- removing any one of the operators would make the set functionally incomplete. **NAND alone is functionally complete; so is NOR alone.**

Flip Flops

Delay flip-flop: takes input and, with some delay, sets output equal to the input.
Requires constant updating to maintain state.
Grabs rising edge input, outputs until next clock cycle.
Current state does not affect next state.
Flip-flops usually produce next state and negation of next state simultaneously.
T Flip-Flop
Toggle flip-flop: if input is 1, toggle current state.
Uses current state to determine next state.
T ≡0⇒"Hold". Next state is same as current.
T ≡1⇒"Toggle". Next state is opposite of current.
Set Reset (sr) Flip-Flop
Two inputs, S (set), R (reset), synchronized by a clock.
S≡1⇒"Set". Next state is 1.
R≡1⇒"Reset". Next state is 0. S≡0∧R≡0⇒"Hold".
Can not have both S and R set to 1
JK Flip-Flop
JK flip-flop -> Two inputs, J (set), K (reset), synchronized by a clock.
Same as SR except with toggle. j≡1∧K≡1⇒"Toggle".

Finite State Machines

FSMs have three components: state, input, output. Clock controls when inputs read ⇒ transitions. PS: present state, NS: next state.
Next state and output is always just some Boolean combination of input and output.
Use our normal 4-step process:
1. Build a truth table,
2. Get canonical form,
3. Simplify,
4. Draw circuit.



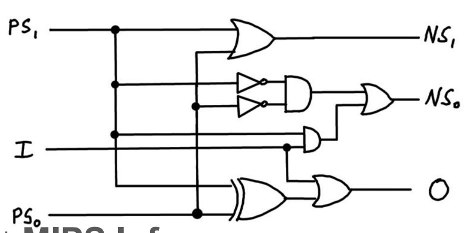
PS	In	NS	Out
00	0	01	0
00	1	01	1
01	-	10	1
10	0	10	1
10	1	11	1
11	0	10	0
11	1	11	1

Let PS = PS_1PS_0, In = I, NS = NS_1NS_0, Out = O.
From the truth table:

$$\begin{aligned}
NS_1 &\equiv \Sigma(2,3,4,5,6,7)\\
&\equiv \overline{PS_1}PS_0\overline{I} + \overline{PS_1}PS_0I + PS_1\overline{PS_0I} + PS_1\overline{PS_0}I + PS_1PS_0\overline{I} + PS_1PS_0I\\
&\equiv \ldots\\
&\equiv PS_1 + PS_0
\end{aligned}$$

$$\begin{aligned}
NS_0 &\equiv \Sigma(0,1,5,7)\\
&\equiv \overline{PS_1PS_0I} + \overline{PS_1PS_0}I + PS_1\overline{PS_0}I + PS_1PS_0I\\
&\equiv \ldots\\
&\equiv \overline{PS_1PS_0} + PS_1I
\end{aligned}$$

$$\begin{aligned}
O &\equiv \Sigma(1,2,3,4,5,7)\\
&\equiv \overline{PS_1PS_0}I + \overline{PS_1}PS_0\overline{I} + \overline{PS_1}PS_0I + PS_1\overline{PS_0I} + PS_1\overline{PS_0}I + PS_1PS_0I\\
&\equiv I + \overline{PS_1}PS_0 + PS_1\overline{PS_0}\\
&\equiv I + (PS_1 \oplus PS_0)
\end{aligned}$$



MIPS Info

Registers: 32 general purpose 32-bit integer registers, denoted $0–$31.
$0 always holds the value 0.
$31 is reserved as the link register: stores the point in instruction memory to return to after a function call.
$PC holds the program counter– address of current instruction $HI & $LO store results of multiplication/division
Memory: 32-bit words and 32-bit memory addresses. **Byte-addressable memory.**
Indexed like a big array of bytes: Mem[0], Mem[1024], Mem[32768].
RTL Examples 3-Operand Arithmetic:
add $8, $9, $10 ≡ R[8] ← R[9] + R[10];
sub $8, $9, $10 ≡ R[8] ← R[9]- R[10];
2-Operand (Immediate Arithmetic):
addi $8, $9, 127 ≡ R[8] ← R[9] + 127;
addi $8, $9, 913 ≡ R[8] ← R[9] + 913; addi $8, $9,-6 ≡ R[8] ← R[9]- 6;
Data Transfer (Memory Accesses):
lw $13, 32($10)≡R[13]<Mem[R[10]+ 32];
sw $13, 8($10) ≡ Mem[R[10] + 8] ←R[13];
3 Operand Arithmetic
op $rd, $rs, $rt ≡ $rd = $rs op $rt
$rd is the destination register.
$rs is the (first) source register.
$rt is the second source register.
op is some arithmetic operation:
add, addu, sub, subu, and, or, xor, ...
2 Operand Arithmetic
op $rt, $rs, imm ≡ $rt = $rs op imma
$rt is the destination register.
$rs is the source register.
imm = immediate - hard-coded value.
op is some arithmetic operations:
addi, addiu, subiu, andi, ori, xori, ...
sll, srl (logical); sla, sra (arithmetic)
(shifts are special case of R-type instr.)
Data Transfer l
w $rt, offset($rs) ≡ $rt = Mem[$rs + offset]
sw $rt, offset($rs) ≡ Mem[$rs + offset]= $rt
$rt is the "value" register.
$rs is the "address" register.
offset is an immediate.
also possible to load +store bytes, halfwords: lb, sb (byte); lwr, swr (least-signficiant halfword); lwl, swl (most-significant halfword).
Endianness
Defined Endianness: ordering of multiple bytes intended to be interpreted together as a single number..
Consider the number: 0xAABBCCDD
Little-Endian:
least-significant byte stored/sent first.
Ordering: 0xDD, 0xCC, 0xBB, 0xAA
Big-Endian:
most-significant byte stored/sent first.
Ordering: 0xAA, 0xBB, 0xCC, 0xDD
MIPS is big-endian. Big-endian conceptually easier but little-endian has performance benefits.

More MIPS

Special Register Names

$zero: the zero-valued register ($0)
$at: reserved for compiler ($1)
$v0, $v1: result values ($2, $3)
$a0- $a3: arguments ($4–$7)
$t0- $t9: temporaries ($8–$15, $24, $25) Can be overwritten by callee
$s0- $s7: saved ($16–$23)
Must be saved/restored by callee
$gp: global pointer for static data ($28)
$sp: stack pointer ($29)
$fp: frame pointer ($30)
$ra: return address ($31)

MIPS Instruction Formats

Every instruction in MIPS is 32-bits. All instructions belong to pre-defined format:
R-Type: "Register" I-Type: "Immediate"
J-Type: "Jump"
Each defines how instruction data broken into "bit-fields" + how they are interpreted in ID.
first 6 bits always =opcode. opcode determines type of instruction and format of remaining bits.

R-Type Instructions -"Register type".
usually have 3 registers as its operands.
General arithmetic operations.
op — the opcode - 6 bits
rs — first source register - 5 bits.
rt — second source register - 5 bits.
rd — destination register - 5 bits.
shamt — shift amount - 5 bits; used for shift instructions, 0 otherwise.
funct — the arithmetic function the ALU should perform - 6 bits.

I-Type Instructions - "Immediate type"
always have 2 registers and an immediate.
op — the opcode - 6 bits.
rs — first source register - 5 bits.
rt — second source/destination register - 5 bits.
imm — the immediate/constant - 16 bits.

J-Type Instructions - Jump type
have just one big immediate, called a target.
Only two instructions: j (jump) and jal (jump and link).
Op - 6 bits, target (jump address) 26 bits- target memory address to jump to.
Note: target is always multiplied by 4 before being applied to program counter

J-Type Instructions and Pseudo-Direct Addressing Pseudo-Direct Addressing:
Almost a direct addressing of instruction memory. Compiler usually handles the calculation of the exact jump target. Next value is target × 4 + upper 4 bits of current PC.
nPC = (PC & 0xf0000000) | (target << 2);

Summary of Control Signals

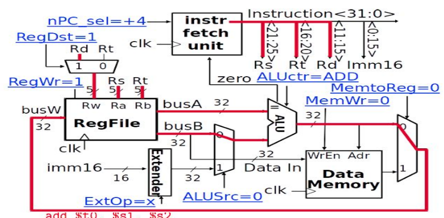
func	10 0000	10 0010	Doesn't Matter				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPC_sel	0	0	0	0	0	1	?
Jump	0	0	0	0	0	0	1
ExtOp	x	x	1	1	1	x	x
ALUctr	Add	Subtract	Or	Add	Add	Equal	x

x = Don't care / Doesn't matter

Note: numeric values not really important. Just gives semantic meaning.

Tracing add in full

`add $rd, $rs, $rt`



`add $t0, $s1, $s2`

op	$s1	$s2	$t0	shamt	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

`sub $t0, $s1, $s2`

op	$s1	$s2	$t0	shamt	sub
0	17	18	8	0	34
000000	10001	10010	01000	00000	100010

`sll $s0, $t0, 4`

op	rs	$t0	$s0	4	shift left
0	0	8	16	4	0
000000	00000	01000	10000	00100	000000

`lw $t1, 12($t0)`

op	rs	rt	immediate
35	$t0	$t1	12
100011	01000	01001	0000000000001100

`sw $t1, 32($t0)`

op	rs	rt	immediate
43	$t0	$t1	32
101011	01000	01001	0000000000100000

`addi $t1, $t0, 10`

op	rs	rt	immediate
8	$t0	$t1	10
001000	01000	01001	0000000000001010

`addiu $t1, $t0, 10`

op	rs	rt	immediate
9	$t0	$t1	10
001001	01000	01001	0000000000001010

unsigned instructions will *not* signal exception on overflow

MIPS ISA: Some Important Instructions

Category	Instruction		OP/ funct	Example	Meaning
Logic & Arith.	add	R	0/32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	R	0/34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	I	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	and/or	R	0/(36/37)	(and/or) $s1, $s2, $s3	$s1 = $s2 (∧/∨) $s3
	(and/or) immediate	I	12/13	(andi/ori) $s1, $s2, 6	$s1 = $s2 (∧/∨) 6
	shift right logical	R	0/2	srl $rd, $rt, 4	$rd = $rt >> 4
	shift right arithmetic	R	0/3	sra $rd, $rt, 4	$rd = $rt >> 4
Data Transfer	load word	I	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	I	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	I	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	I	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
Cond. Branch	br on equal	I	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	I	5	bne $s1, $s2, L	if ($s1 != $s2) go to L
	set less than	R	0/42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set less than immediate	I	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump	jump	J	2	j 250	go to 1000
	jump register	R	0/8	jr $t1	go to $t1
	jump and link	J	3	jal 250	go to 1000; $ra=PC+4

Note: knowing the binary values of each bit-field is not neccesary, but understanding the semantic meaning of each instruction *is* important.

Full Method Example: C to MIPS

```
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:  sll $t1, $a1, 2     # $t1 = k * 4
       add $t1, $a0, $t1   # $t1 = v+(k*4)
                           # (address of v[k])
       lw $t0, 0($t1)      # $t0 (temp) = v[k]
       lw $t2, 4($t1)      # $t2 = v[k+1]
       sw $t2, 0($t1)      # v[k] = $t2 (v[k+1])
       sw $t0, 4($t1)      # v[k+1] = $t0 (temp)
       jr $ra              # return to calling routine
```

Note: words and int-type are both 32-bits here.

MESI Protocol

an invalidate snooping protocol which adds several optimizations to further reduce bus bandwidth.
If cache block is "Exclusive" avoid broadcasting a write.
MESI can be described by looking at what happens on read misses, read hits, write misses, and write hits.
Modified: The cache block is exclusively owned by the processor and is dirty (i.e. it differs from main memory).
Exclusive: exclusively owned by the processor and is clean (i.e. the same value as main memory).
Shared: shared between multiple processors, clean.
Invalid: loaded into cache but value no longer valid. .

MESI Read Hit
If cache = MES, return the value, no state change.

MESI Read Miss
Case 1: No copies anywhere. response from lower level memory. value stored in cache with state E.
Case 2: One cache has an E copy. cache hears read request, puts copy of value on bus. Both cores set cache block's state to S.
Case 3: Several caches have an S copy. One snooping cache hears request, puts copy on bus. requesting core reads value, sets state to S. All copies in state S.
Case 4: One cache has an M copy. The snooping cache hears read request, puts copy on bus. The requesting core reads value, sets state to S. snooping sets state to S, writes updated value to main memory.

MESI Write Hit
If in M state: Cache block is exclusively owned and already dirty. Update cache value, no state change. **If in E state:** Cache block is exclusively owned and clean. Update cache value, set state to M. **If in S state:** Cache block is shared but clean. Requesting core broadcasts invalidate on the bus. Snooping cores set to I. Requesting updates cache, sets state to M.

MESI Write Miss Need to read first and then write.
Case 1: No other copies. Read from main memory, write new value in cache, set state to M.
Case 2: Other copy is in E state / Other copies are in S state. The requesting cache requests **Read With Intent To Modify**. The snooping cache(s) hear **RWITM** request, puts copy on bus, sets own state to I. Requester reads value, updates value, sets state to M.
Case 3: Other copy in M state The snooping cache hears RWITM, puts copy on bus, writes back to main memory, sets its own state to I. Requester reads value from bus, updates value, sets state to M.

Pipelining

technique for instruction-level parallelism where each stage of datapath is kept busy. Instructions overlapped. Each stage executes different instruction

Single cycle datapath:
Minimum clock cycle = sum of all stages since some instructions (load word) use all stages

Multi-cycle datapath:
Minimum clock cycle = long enough for slowest stage in the pipeline

Quantifying Pipelined Speedup
If the time for each stage is the same:
Ideal Speedup = Number of Stages
If the time for each stage is not the same: Ideal Speedup = Time between instructions (single-cycle) / Time between instructions(pipelined)
Actual Speedup = Time to complete(single-cycle) / Time to complete(pipelined)
Note that a single-cycle datapath always has latency less than or equal to a multi-cycle/pipelined datapath.

Calculating Speedup
Single-cycle datapath: 800ps clockcycle.
Pipelined: 200ps clockcycle.
Uneven time for each stage.
ID and WB only100ps. Given 3 lw instructions.
-> IdealSpeedup= 800/ 200 =4
Actual Speedup= 2400/1400 =1.714
If we have many instructions?
Actua lSpeedup= 1000000×800 / (1000000×200+800) ≈4

Calculating Pipelined Time
Classic Performance Equation:
CPU time = Instruction_count ×CPI×clock cycle
Time for pipelined execution:
Time pipelined = Fill time + IC ×clock cycle) (Assuming no stalls or hazards.) Once pipeline is full, one instr. completes every cycle ⇒ CPI is 1. -> Gives IC × 1 × clock cycle
Pipeline only not full during fill\ drain time. Fill time = Drain time = (number of stages- 1) × clock cycle, Assuming number of instructions > number of stages.

Pipelining 2 - Hazards

Structural hazards caused by hardware access conflicts
-> Register access fast, can do twice in 1 clock cycle.
-> Banked L1 for simultaneous instruction / data access.
Data hazards caused by Read After Write (RAW).
Solve = ALU-ALU forwarding. MEM-MEM forwarding
Load-use hazard: stall (load-delay slot), MEM-ALU forward.
Control hazards caused by branch instructions.
Special branch comparator in ID stage.
Branch delay slot; delayed branching.
Branch prediction and pipeline flush. Compiler handles nop insertion to fix hazards. Hardware handles fixing hazards with pipeline interlock.

VLIW Processors

VLIW processors have very long instruction words. Essentially, multiple instructions are encoded within a single (long) instruction memory word called an issue packet. Usually only one lw/sw, only one branch, rest arithmetic. In this case instructions word size ≠ data memory word size.
Simplest scheme: just concatenate multiple instructions together. Ex: Two 32-bit instrs. together in a single 64-bit instruction word.
In a VLIW pipeline: One IF unit fetches single long word encoding multiple instrs. ID stage must handle multiple simultaneous decodes and register reads. In EX stage, each instr. is issued to a different execution unit (ALU). Only one data memory to read/write from!

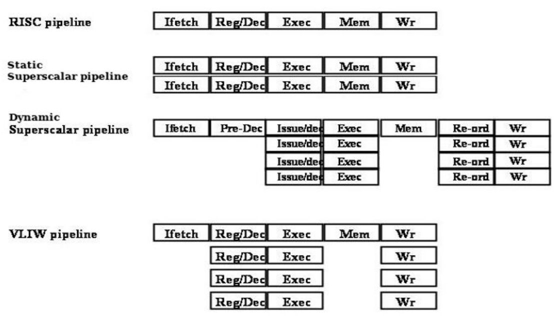
A VLIW Example (3/3)

```
loop:  lw $t0, 0($s1)       # $t0=array element
       addu $t0, $t0, $s2   # add scalar in $s2
       sw $t0, 0($s1)       # store result
       addi $s1, $s1, -4    # decrement pointer
       bne $s1, $0, loop    # branch if $s1 != 0
```

	ALU or branch	Data transfer	CC
loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $0, loop	sw $t0, 4($s1)	4

- Scheduled 5 instructions in 4 cycles. In the limit, CPI approaches 0.8.
- nops don't count towards performance.
- Sometimes when scheduling code you need to adjust offsets.

Comparing Multiple Issue Pipelines (2/2)



VLIW
- Static scheduling.
- In-order execution.
- Single IF unit but many EX units.
- Instructions packed together in issue packet by compiler.

Static SS
- Static scheduling.
- In-order execution.
- Many IF units (or one IF fetching multiple instr.) and many EX units.
- Compiler explicitly schedules each "route" through the datapath.

Dynamic SS
- Dynamic scheduling
- Out-of-order exec.
- Single IF unit but many EX units.
- IF unit might fetch multiple instr. per cycle.

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
add $t6, $t0, $t2				V4	add V4, V0, V1
sub $t4, $t2, $t0			V5		sub V5, V1, V0
xor $t0, $t6, $t2	V6				xor V6, V4, V1
and $t2, $t2, $t6					add ??, V1, ??

Instr.	$t0	$t2	$t4	$t6	Renamed Instr.
Initially	V0	V1	V2	V3	—
add $t6, $t0, $t2				V4	add V4, V0, V1
sub $t4, $t2, $t0			V5		sub V5, V1, V0
xor $t0, $t6, $t2	V6				xor V6, V4, V1
and $t2, $t2, $t6		V7			add V7, V1, V4

True sharing caused by communication of data between threads.
- Data is truly invalided and must be read again.
- Would still occur if cache block size was 1 word.

False sharing caused by multiple processors writing to different memory addresses within a single cache block.
- The cache block is shared but no word within the block is actually shared.
- Miss would **not** occur if cache block size was 1 word.
- Invalidation on each write causes a cache miss for the other processor.
- Goes back and forth in this way ⇒ thrashing.

Let x1 and x2 belong in the same cache block. Processor 1 and Processor 2 each want to access either x1 or x2. Let us assume both processors begin with the cache block loaded into cache.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		Hit; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		Hit; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1