CPSC 330
Computer Organization and Design

FINAL REVIEW

What to expect

- True/False questions (about 20)
- Review Cache memory
- Be able to calculate the size of a cache.
- Calculate performance of a processor; execution time, CPI, clock rate.
- Improving cache performance (AMAT)
- Data hazards, NOPS, Data forwarding, Pipelining
- No questions on virtual memory.
- Review past HW assignments and exams
- MIPS Assembly instructions

How to improve performance

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}} = \frac{\text{cycles/program}}{\text{clock rate}}$$

- So, to improve performance (everything else being equal) one can either

  ↓ the # of required cycles for a program, or

  ↓ the clock cycle time or, said another way,

  ↑ the clock rate.

Performance
CPI example 1

- Suppose we have two implementations of the same instruction set architecture (ISA). For some program,

  Machine A has a clock rate of 4 GHz and a CPI of 2.0.
  Machine B has a clock rate of 2 GHz and a CPI of 1.2.

- *What machine is faster for this program, and by how much?*

Performance
CPI example 2

Two computers, C1 and C2, have the following metrics when running the same program. Which computer is faster? Which has higher MIPS?

	Computer C1	Computer C2
Instructions #	10 billion	8 billion
Clock Rate	4 GHz	4 GHz
CPI	1.0	1.1

Performance
In terms of instruction count…

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

Where "C" is the count of the number of instructions of class "i" executed and "n" is the number of different instruction classes.

Performance *example*
In terms of instruction count…

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three CPI (respectively).
- The first code sequence has 5 instructions:
  - 2 of A, 1 of B, and 2 of C
- The second sequence has 6 instructions:
  - 4 of A, 1 of B, and 1 of C.
- *Which sequence will be faster? How much? What is the CPI for each sequence?*

Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

	Instruction_ count	CPI	clock_cycle
Algorithm	**X**	**X**	
Programming language	**X**	**X**	
Compiler	**X**	**X**	
ISA	**X**	**X**	**X**
Technology			**X**

Example (continued)

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			$\sum =$ 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

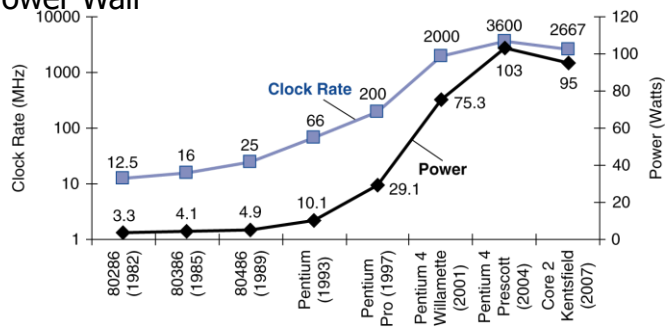
  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to shave a cycle off the branch time?

  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?

  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Power Trends

The Power Wall



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination is first)

Example:

C code: `a = b + c`
MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

- **Translation from C to MIPS is performed by a __compiler_____**

Compiling a C assignment using Registers

- `f = (g + h) - (i + j)`

The variables f, g, h, i and j are assigned to registers $s0, $s1, $s2, $s3, and $s4 respectively.

$t0, $t1 → temporary registers

What is the compiled MIPS code?

Instructions

- Load instruction
- Example:
  Assume A is an array of 100 words. Variables g and h are associated with reg **`$s1`** and **`$s2`**. Starting (base) address of the array is in **`$s3`**.

C code: **`g = h + A[8];`**

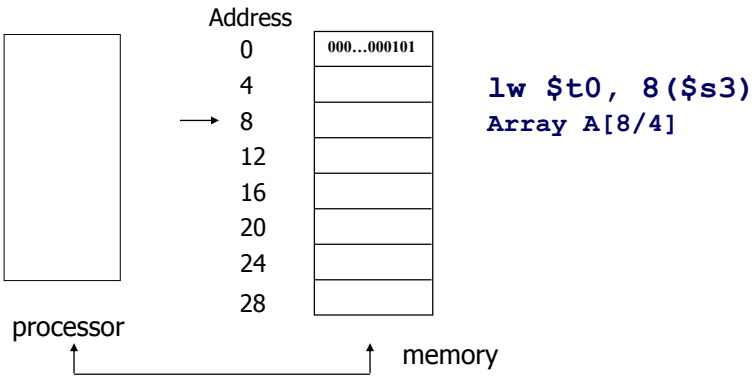
Operand in memory

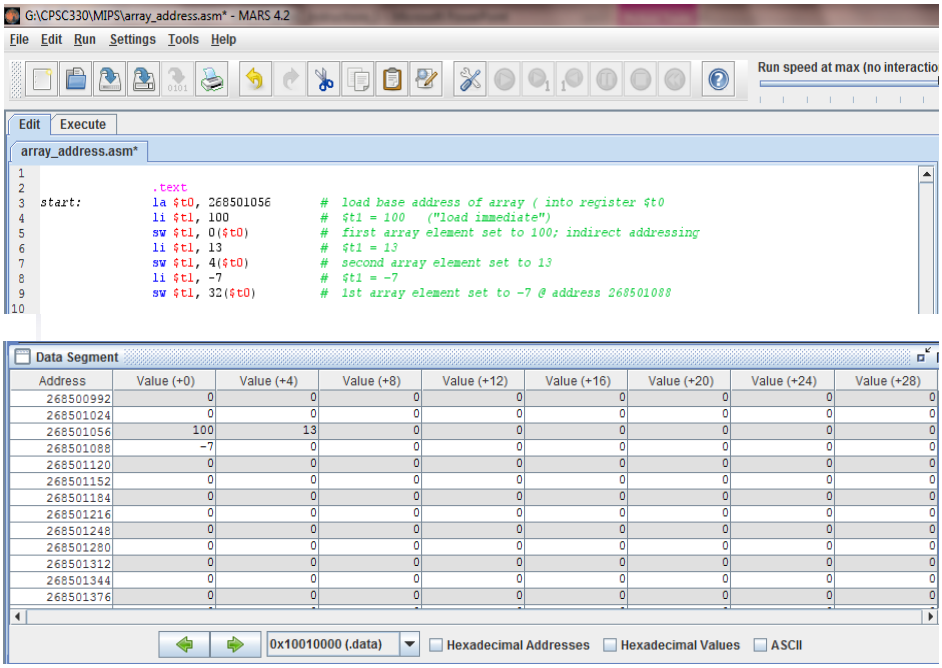
MIPS code:

```
lw $t0, 32($s3)     # temp. reg $t0 = A[8]
add $s1, $s2, $t0   # put sum in reg corresponding to g
                    # $s1 = $s2 + $t0 → g = h + A[8]
```

Addressing: Byte vs. Word

- Most data items use larger "words"
- Today, machines address memory as bytes, hence word addresses differ by 4
- For MIPS, a word is 32 bits or 4 bytes
- In MIPS, words must start at addresses that are multiples of 4.



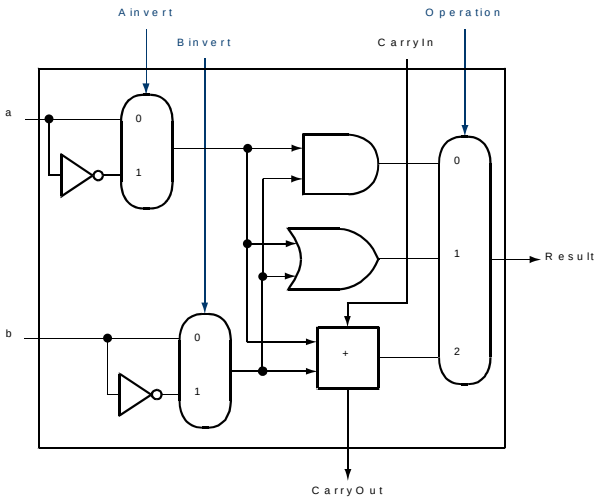


MIPS Instructions - Review

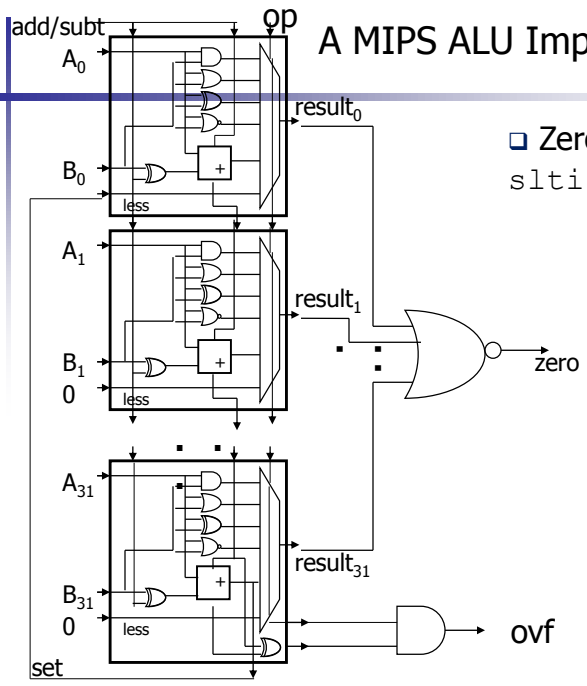
Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`
`addi $s1,$s1,4`	`$s1 = $s1 + 4`
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`
`slt $t0,$s0,$s1`	`if ($s0 < $s1) then $t0 gets 1 otherwise $t0 gets 0`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100/4]`
`sw $s1,100($s2)`	`Memory[$s2+100/4] = $s1`
`bne $s4,$s5,L`	`Next instr. is at Label if $s4 ≠ $s5`
`beq $s4,$s5,L`	`Next instr. is at Label if $s4 = $s5`
`jr $t1`	`jump via register: go to the address specified by $t1`
`j Label`	`go to the target address`
`mult $t1,$t2`	**`{Hi,Lo} = $t1*$t2`**
`div $t1,$t2`	**`Lo = $t1/$t2; Hi = $t1%$t2`**
`mflo $a0`	**`move from Lo to $s0`**

new

ALU



A MIPS ALU Implementation

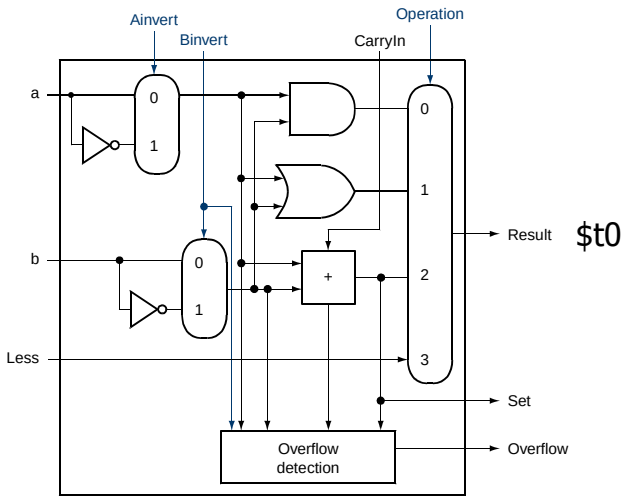


- Zero detect (`slt, slti, beq, bne`)

Tailoring the ALU to the MIPS

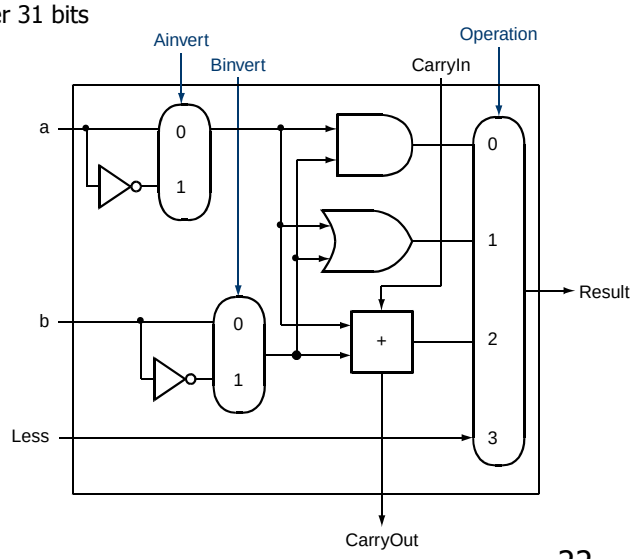
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - slt $t0, $t1, $t2
  - produces a 1 if $t1 < $t2 and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
  - remember subtraction: a + ($\overline{b}$ +1)
- Need to support test for equality.
  - beq $t1, $t2, branch target address.
  - use subtraction: (a-b) = 0 implies a = b

Supporting 'slt'
slt $t0, $t1, $t2



Supporting 'slt' (continued)

all other 31 bits

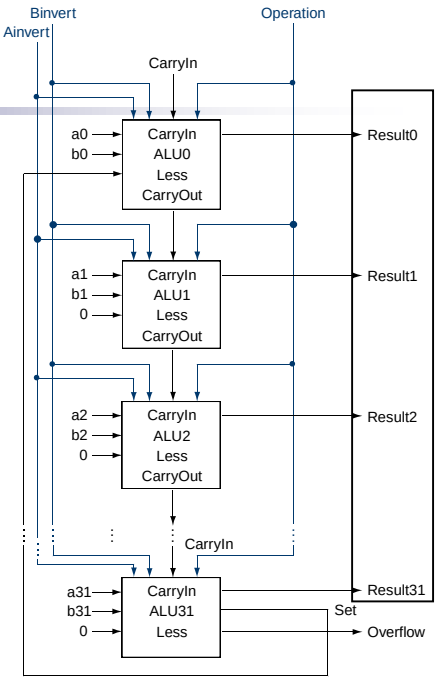


Supporting 'slt'

- If the ALU performs a – b (select 3 in the MUX), then the result = 0…0001 if a < b → slt sets LSB to 1 and all other higher order bits to 0.
- If the ALU performs a – b, then the result = 0…0000 if a > b → All bits are set to 0.

(a) Try: a=0...0011110, b=0...0101110
Result → $t0 = 0...001

(b) Try: a=0...0101110, b=0...0011110
Result → $t0 = 0...000

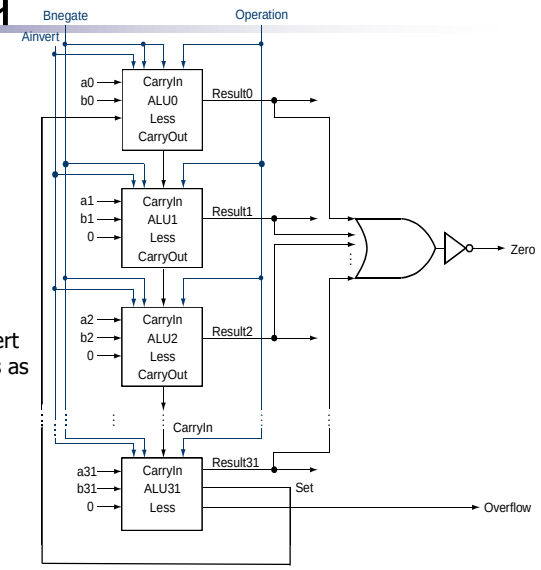


Supporting 'beq'

- To check for equality:
  a – b = 0 → a = b
- *'Zero' is a ____ when the result is zero.*
- IF a – b ≠ 0 → a ≠ b
- *'Zero' is a ____ when the result is not zero.*

We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit MUX Operation lines as 4- bit control lines for the ALU

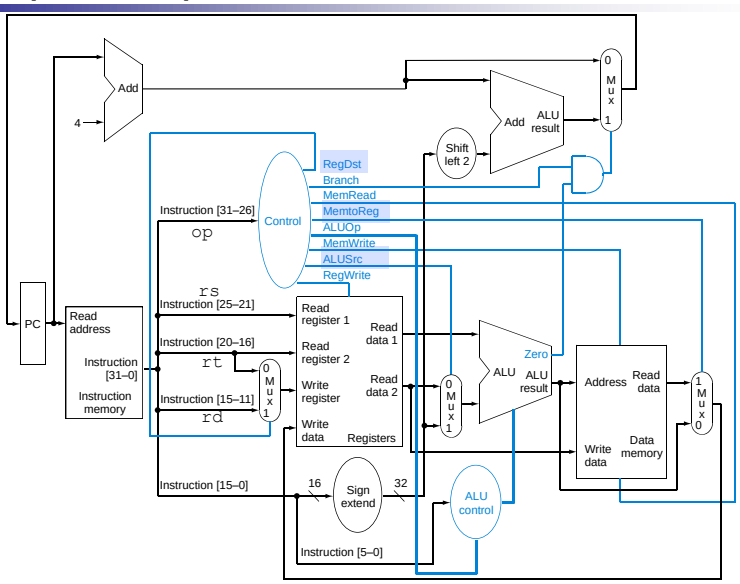
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



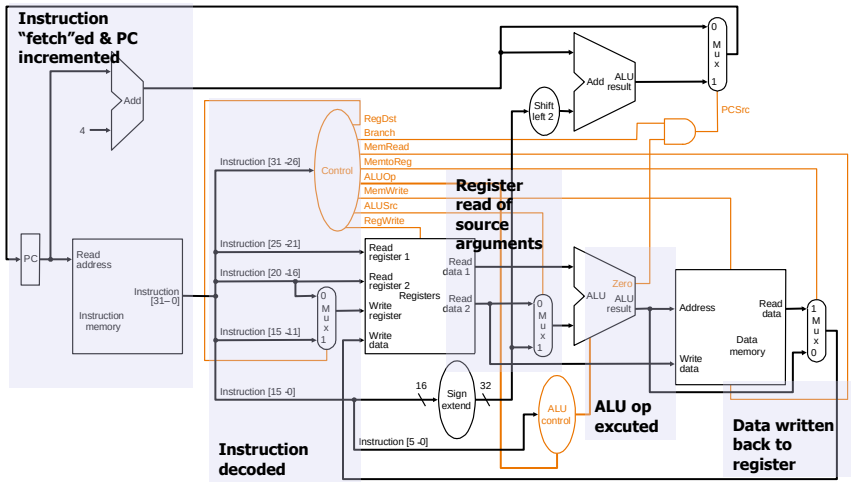
$$Zero = \overline{(Result31 + Result30 + \ldots + Result2 + Result1 + Result0)}$$

24

Simple datapath with control unit

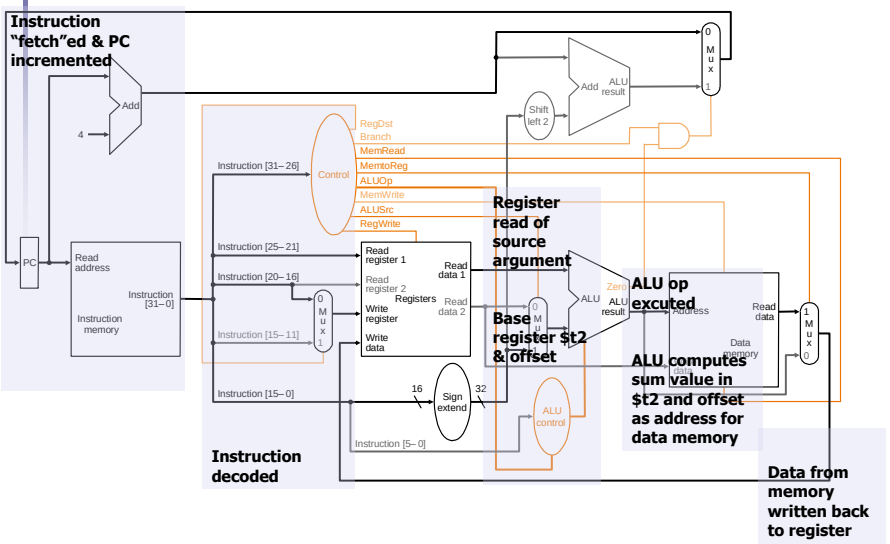


Datapath & Control
phases of R-type instruction

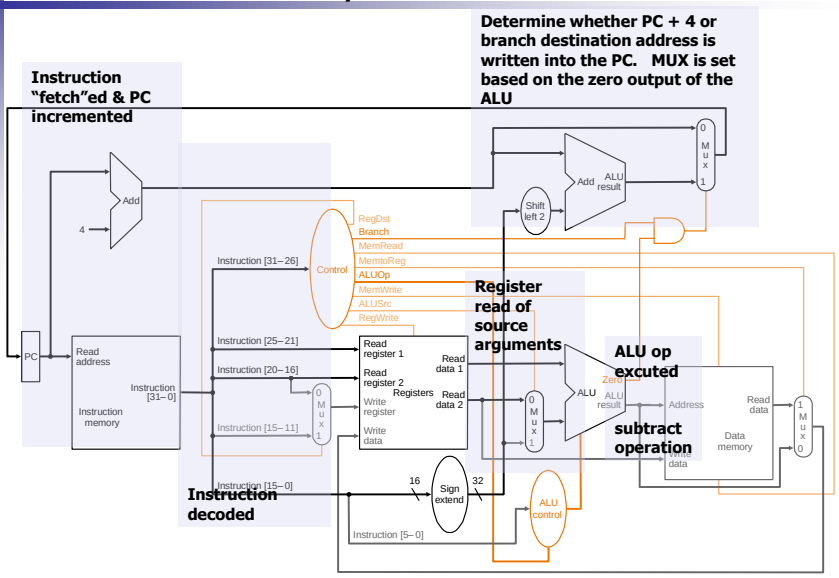


Datapath & Control
flow of 'load' instruction

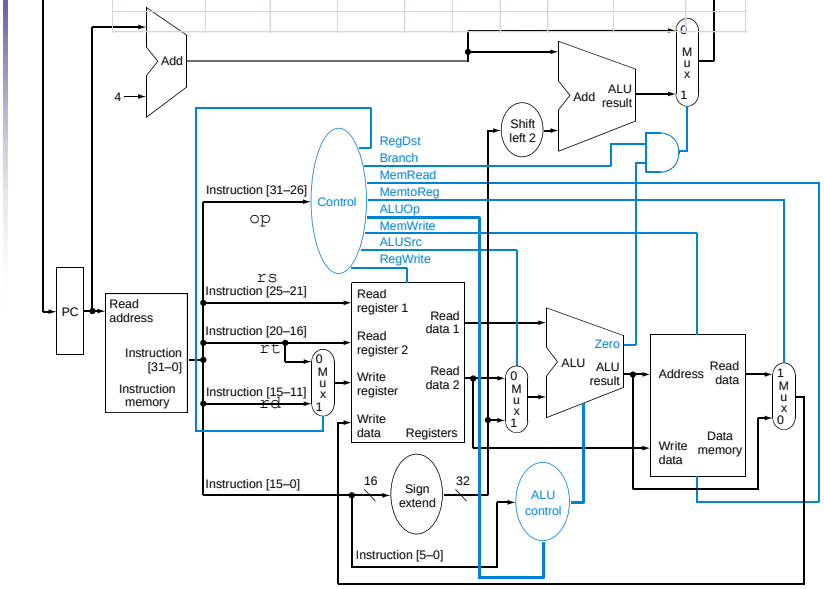
lw $t1, offset($t2)



Datapath & Control
flow of 'branch equal' instruction



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format									

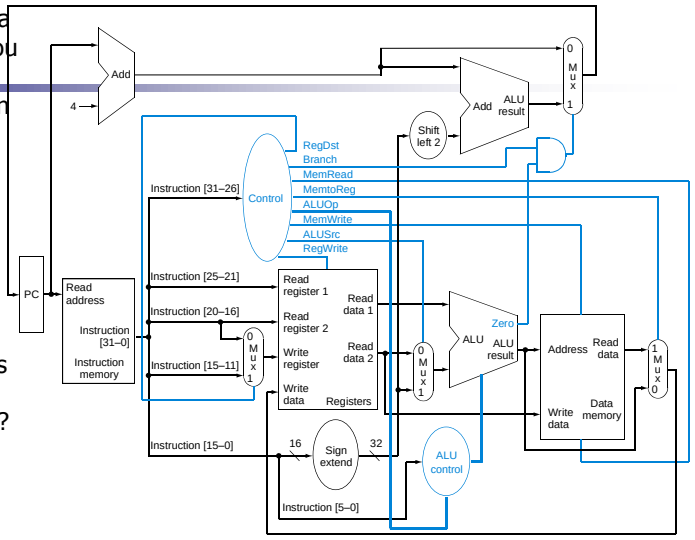


You were testing a datapath when you discovered the following 'stuck on 0' faults:

- RegWrite = 0
- ALUop1 = 0
- ALUop0 = 0
- Branch = 0
- MemRead = 0
- MemWrite = 0

Which instructions will not work correctly? Explain?

ALUOp	Operation
00	lw
00	sw
01	beq
10	add
10	sub
10	AND
10	OR
10	slt



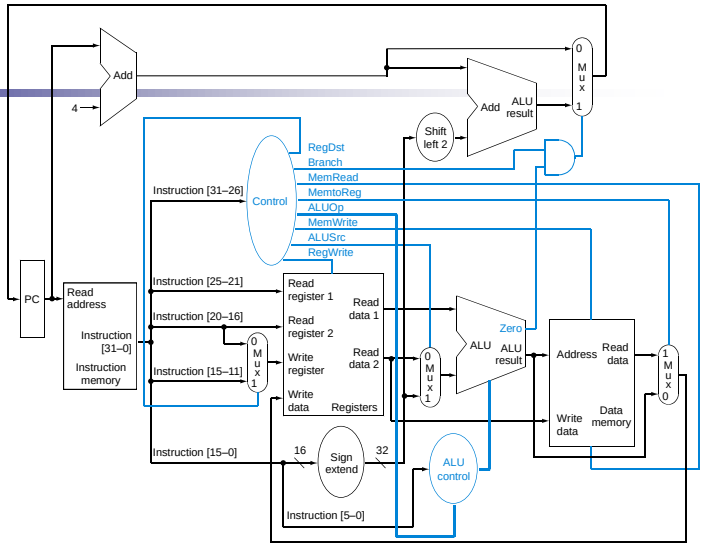
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Consider the following faults:

- RegWrite = 1
- ALUop0 = 1
- ALUop1 = 1
- Branch = 1
- MemRead = 1
- MemWrite = 1

ALUOp	Operation
00	lw
00	sw
01	beq
10	add
10	sub
10	AND
10	OR
10	slt

Which instruction(s) will not work correctly?

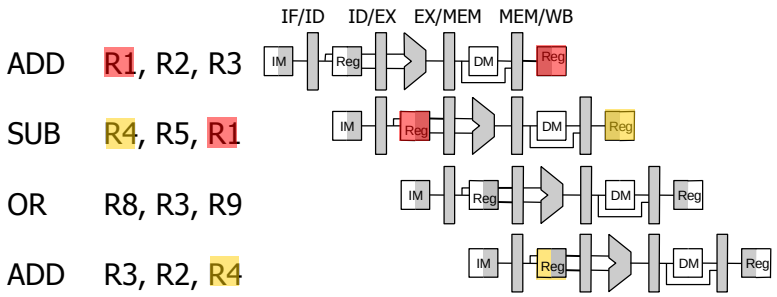


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Hazards
3 types of pipelining hazards

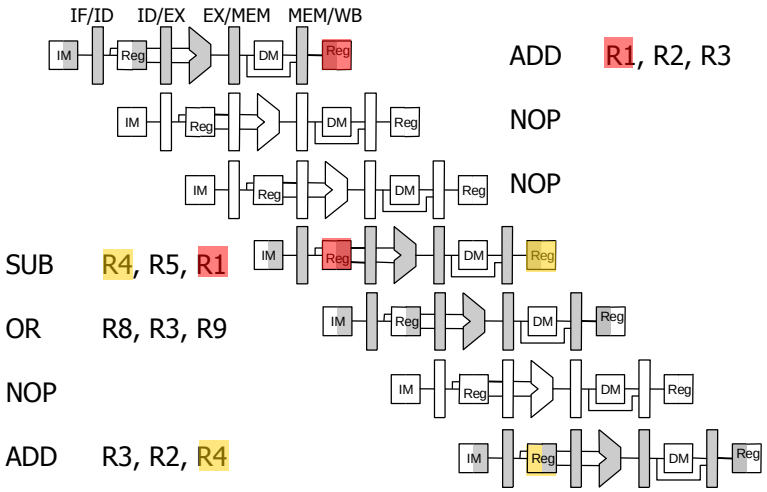
- structural hazards: attempt to use the same resource two different ways at the same time
  - A register is used to write back to at the same time the same register is used to read/ 'put bits' into from the current instruction in the decode cycle.
- data hazards: attempt to use a register before its proper value is ready.
  - instruction depends on result of prior instruction still in pipeline
- control hazards: attempt to make a decision but the information needed to make the decision is not available yet.
  - Branch instructions

Data Hazard - *Problem*



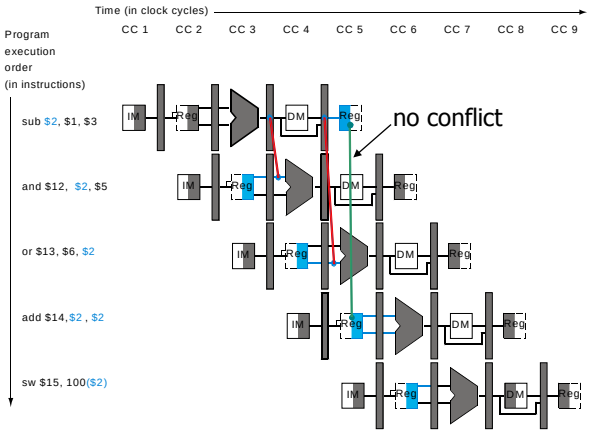
- ❖ Identify the instruction(s) affected by the data hazard.
- ❖ Fix the hazard by inserting 'nops' ← note: not efficient

Fixing the data hazard using *software: 'NOPS'*

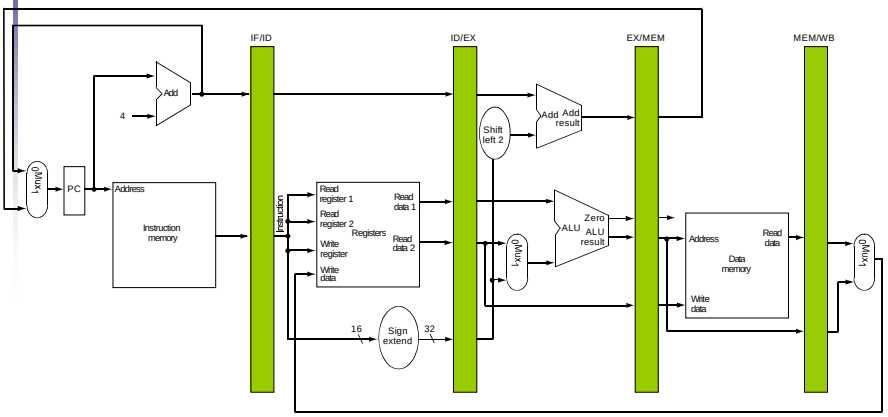


Fixing the data hazard using *data forwarding*

- Use temporary results, don't wait for them to be written
- Pipeline registers have data too!
- "sub" ALU operation completes prior to "and" execution
- Same for "or"
- No conflict with "add", since write completes before read in same cycle



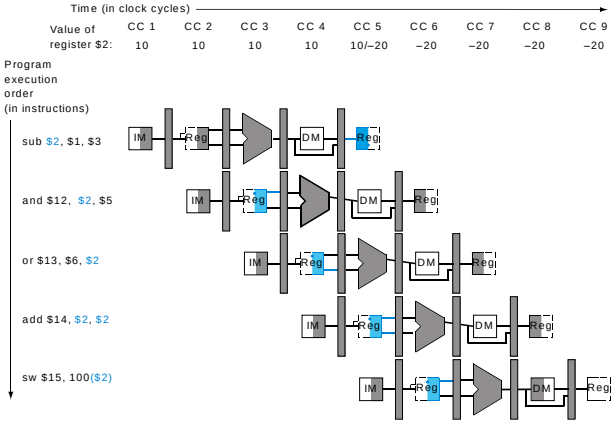
Pipelined Datapath



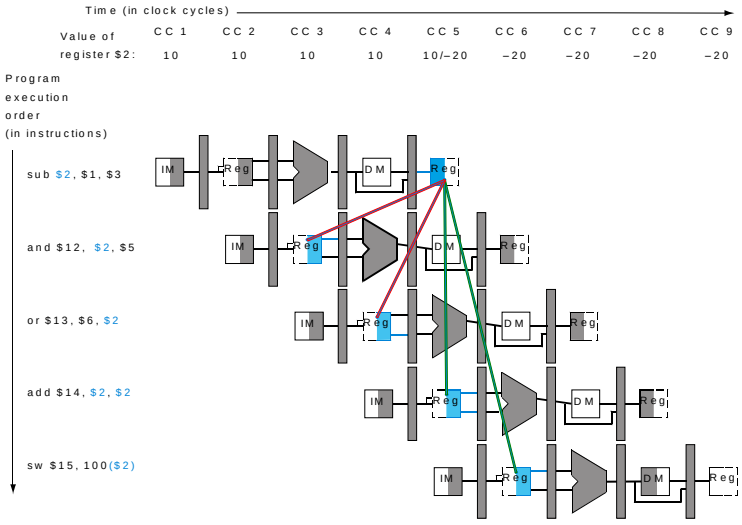
- In between stages we need to store "results"
- So we add "pipeline registers" between stages

Dependencies

- Problem with starting next instruction before first is finished
  - Dependencies that "go backward in time" are data hazards
  - Can we identify the data hazard?



Pipeline Data Hazards



Software Solution

- Have compiler guarantee no hazards
- Where do we insert the "nops" ?

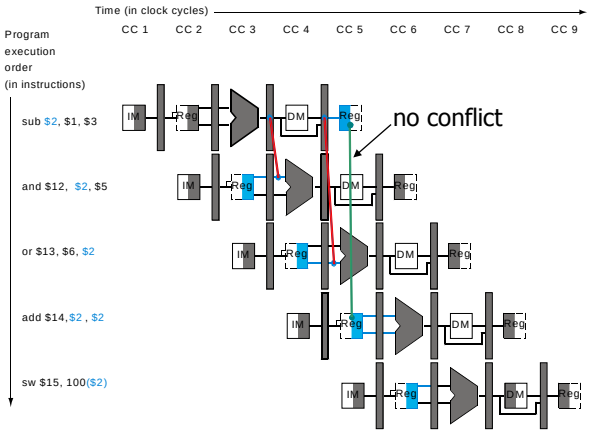
```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

Data Hazards
"It's a software problem!"

```
sub $2, $1, $3    # Reg $2 written with "sub" result
nop               # Do nothing for a cycle
nop               # Do nothing for another cycle
and $12, $2, $5   # 1st op depends on "sub" result
or  $13, $6, $2   # 2nd op depends on "sub" result
add $14, $2, $2   # Both ops depend on "sub" result
sw  $15, 100($2)  # Dest addr depends on "sub" result
```

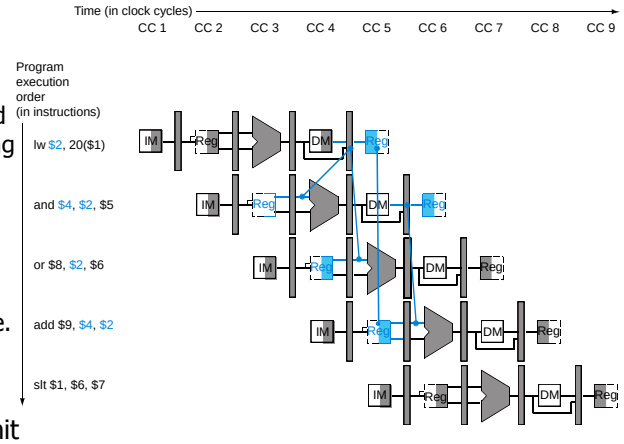
Fixing the data hazard using
data forwarding

- Use temporary results, don't wait for them to be written
- Pipeline registers have data too!



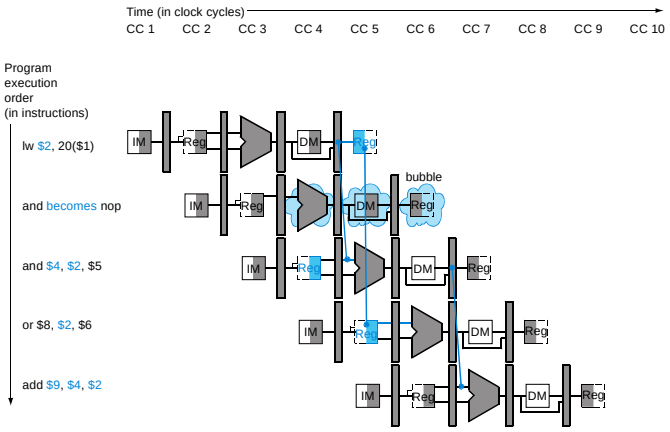
Can't always forward

- Load word can still cause a hazard:
  - an instruction (and) tries to read a register following a load instruction that writes to the same register.
  - 'lw' & 'and' dependence goes backwards in time.
- Thus, we need a hazard detection unit to "stall" the load instruction

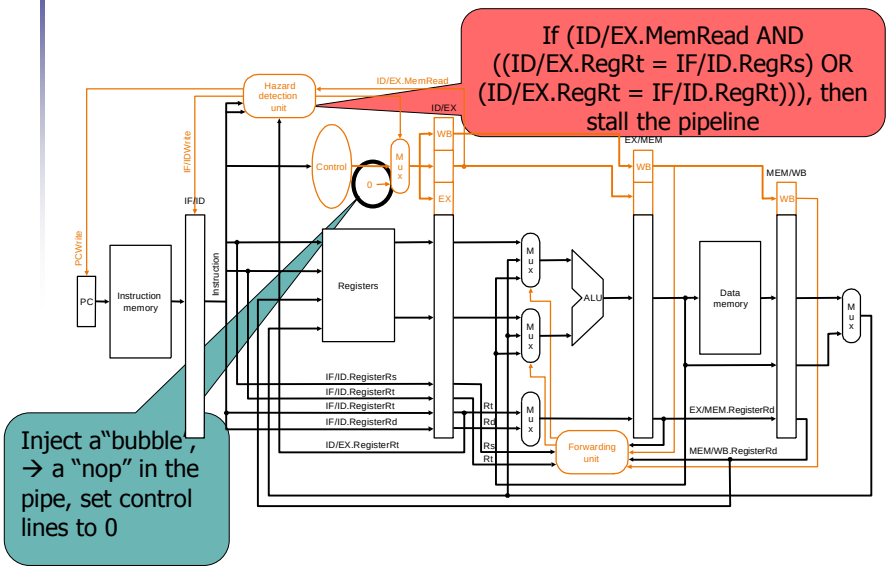


Data Hazards *and "stalls"*

- By stalling instructions in the pipe 1 cycle, dependence is gone
- A "stall" is said to inject a "bubble" or "nop" into the pipe



Detecting "lw" hazard
& injecting a "bubble" (nop)

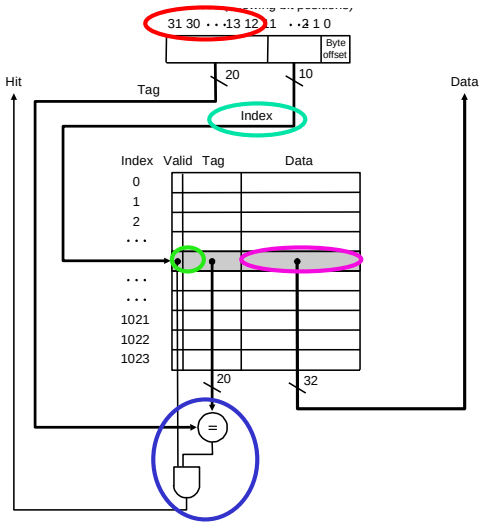


Memory Hierarchy
Locality of Reference

- Principle that most programs spend time in tight loops or working on the same data repeatedly
- *Temporal Locality* : the tendency to reuse recently accessed data
  - Reason for algorithms such as "most recently used"
- *Spatial Locality* : the tendency to reference data that is "close" to other data (nearby physical addresses)
  - Reason for moving "blocks" of memory at at time

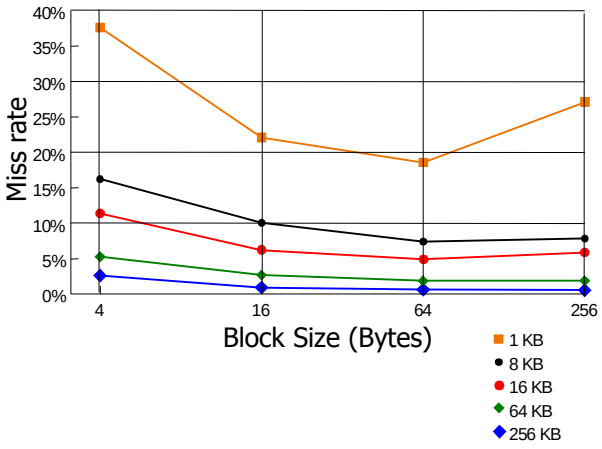
Cache Basics
anatomy of direct mapped cache

- 4KB cache: 1024 entries, 1 word per block, 4 bytes/word, 2^{12} bytes
- Cache index: indicates this specific cache entry
- Valid bit: indicates whether or not data for this cache entry is valid,
- Tag: identifies which memory location this data block represents (upper bits of data address, for 2^n size cache, these are bits <31..n>)
- Data: cache block (size is implementation dependent)
- Hit: *tag valid & = upper addr bits*



Cache Alternatives
miss rate versus block size

- Increasing block size helps reduce miss rate… up to a point (but then miss penalty takes over). Spatial locality among words in a block decreases with a very large block.
- Increasing size of cache always helps reduce miss rate!



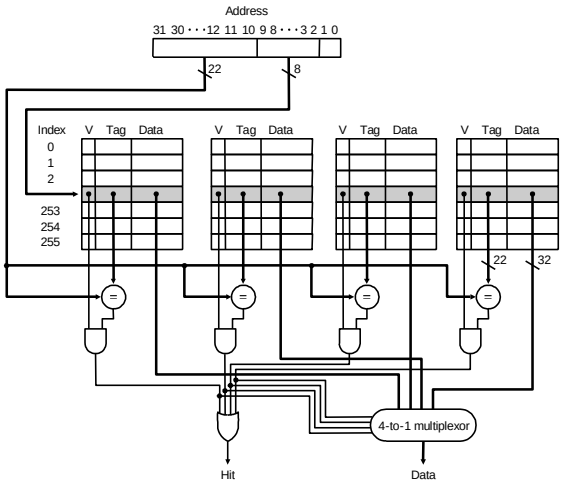
Improving cache performance

Two ways of improving performance:

- Decrease the miss ratio: More flexible placement of blocks → Associativity
- Decrease the miss penalty: Multi-level caching used in high-end computers.

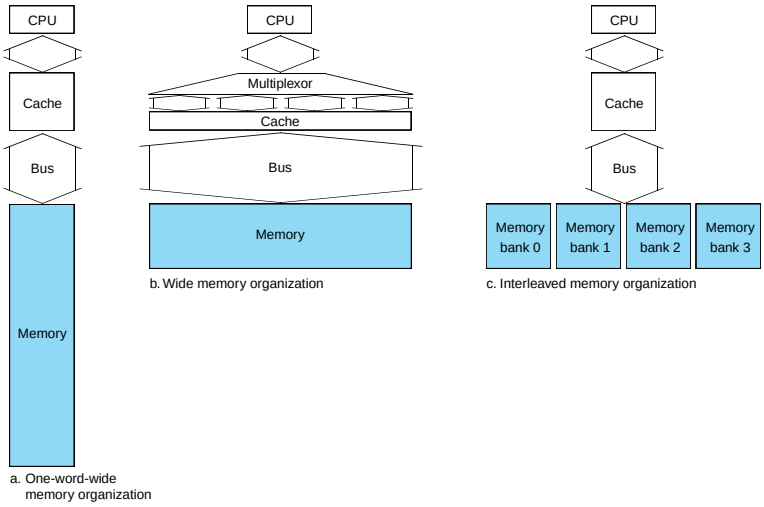
4-Way Set Associative Cache

- 4 block/set becomes the number of simultaneous compares to perform the search in parallel
- Although larger sets increase the probability of a hit, they do so at the expense of more hardware, and consequently access time



Designing the Memory System to Support Caches

- Make reading multiple words easier by using banks of memory



Measuring cache performance

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory\_stall clock cycles}) \times \text{Clock cycle time}$$

$$\text{Memory\_stall clock cycles} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Calculating cache performance - Example

- Assume an instruction cache miss rate for a program is 2% and a data cache miss rate is 4%. If the processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never misses. The frequency of all loads and stores is 36% for SPECint2000.

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory\_stall clock cycles}) \times \text{Clock cycle time}$$

$$\text{Memory\_stall clock cycles} = \frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Improving cache performance

Two ways of improving performance:

- Decrease the miss ratio: More flexible placement of blocks → Associativity
- Decrease the miss penalty: Multi-level caching used in high-end computers selling for more than $10,000 in1990. Today, multi-level caching is common for less than $200 in desktop computers.

Reducing cache misses by more flexible placement of blocks

I. Direct-Mapped cache structure: there's a direct mapping from any block address in memory to a single location in the upper level of the hierarchy.

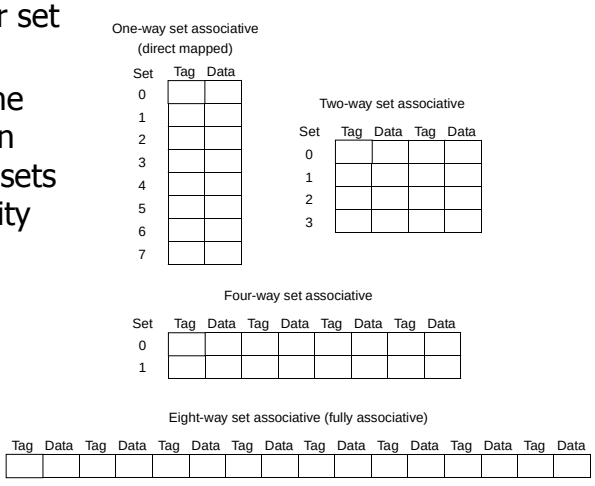
II. Fully Associative cache structure: A block in memory may be associated with any entry in the cache.

III. Set-Associative cache structure: There is a fixed number of locations (at least two) where each block can be placed. Combines direct-mapped and fully associative placement.

Decreasing miss ratio with associativity

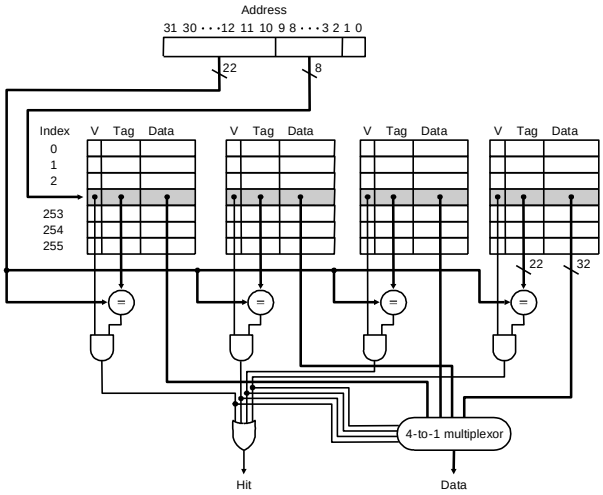
- Increasing the associativity increases the number of _____ per set

Note that the cache size in blocks = # sets x associativity



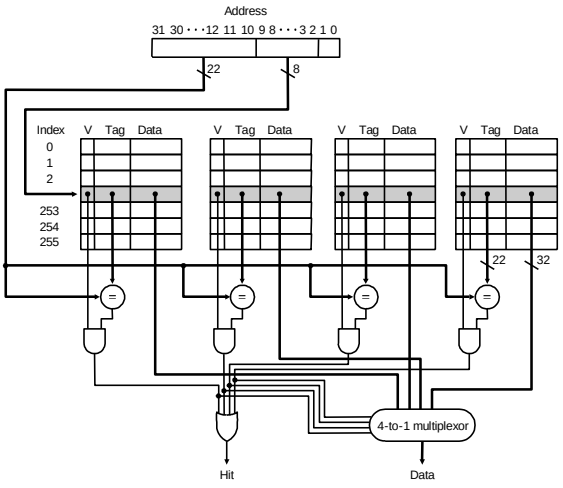
4-way set associative cache

- What is the size of this associative cache in KB?
- Comparators determine _____?
- The output of the comparators (AND gates) is used to _____?



4-Way Set Associative Cache - *continued*

- 4 block/set becomes the number of simultaneous compares to perform the search in parallel
- Although larger sets increase the probability of a hit, they do so at the expense of _____?



MIPS

- *Scoreboard* operations by single-stepping through a program and recording results in trace table.
- Debug/complete one MIPS code. Review MIPS assignments.