Throughput = events per unit time (bandwidth, IPC)
Latency (pipeline) = the # of stages in a pipeline or the # of stages between two instructions during execution
Pipelining = an implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.
· Speedup due to pipelining = # of stages in the pipeline

MIPS Instructions take 5 steps:
1) Fetch 2) Read 3) Execute 4) Access 5) Write
2 : 1 : 2 : 2 : 1

Time between $instr_p$ = $\frac{\text{Time between instruction unpipelined}}{\text{\# of pipe stages}}$

given a large # of instr. speed-up ≈ # of pipe stages

Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction

Hazards

structural hazard = when a planned instruction cannot execute in the proper clock cycle b/c the hardware does not support the combination of instructions that are set to execute

data hazard = when a planned instruction cannot execute in the proper clock cycle b/c data that is needed to execute the instruction is not yet available (stall is required)

forwarding = bypassing = a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory

load-use data hazard = a specific form of a data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction

pipeline stall = bubble = a stall initiated in order to resolve a hazard

Rearrange to Improve Performance:

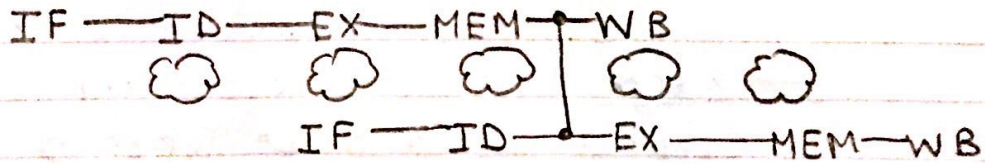
```
$t0:
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

(bypassing fixes; bypass; bypass)

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

(bypass; bypass; bypass)

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```



Control Hazard = Branch Hazard = when the proper instruction cannot execute in the proper pipeline clock cycle b/c the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected

2 solutions to control hazard:

1) Stall = stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from
(resolve branch in 2nd stage of pipeline w/ extra hardware)

2) Predict = always predict branches will be untaken. When you're right, the pipeline proceeds @ full speed.
Only when you're wrong does pipeline stall.

branch prediction = a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Dynamic prediction = may change predictions for a branch over the life of a program
ex: keep a history for each branch as taken or untaken, use the recent past behavior to predict the future

3) Delayed Decision = delayed branch always executes next sequential instruction, w/ the branch instr. taking place after that 1 instr. delay

five components of MIPS instructions flow left-to-right except for:

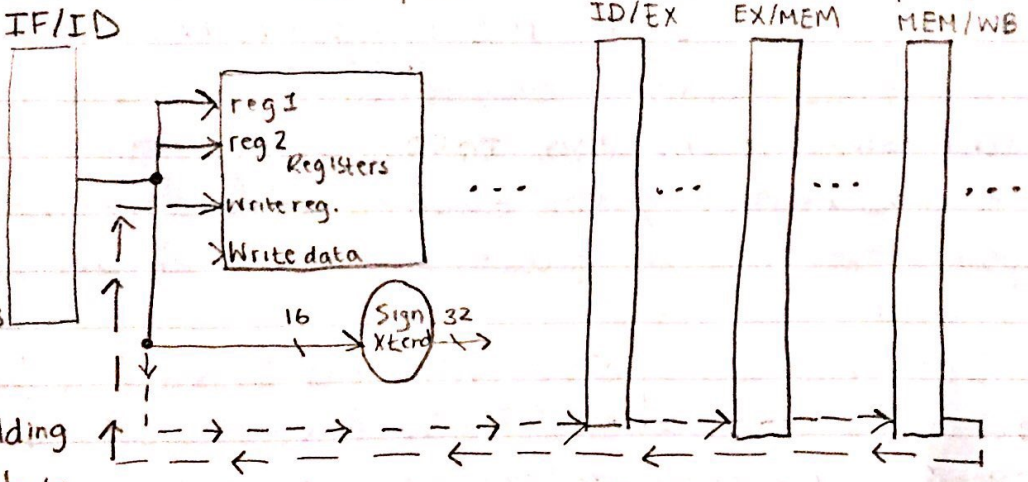
- write-back stage, which places result back into the register file in the middle of the data path
- the selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

pipeline register must be 64 bits wide b/c it must hold the 32-bit instr. fetched & the incremented 32-bit PC address

ID/EX = 128 bits
EX/MEM = 97 bits
MEM/WB = 64 bits

corrected pipelined datapath to handle lw properly:

The reg # is passed from the ID stage until it reaches the MEM/WB pipeline register, adding five more bits to the last 3 pipeline registers.



now a days it's just called this

Exception = interrupt = an unscheduled event that disrupts program execution; used to detect overflow

Interrupt = an exception that comes from outside the processor

EX: I/O device request => External => Interrupt
Invoke op sys. from user program => In => Exception.
Arithmetic Overflow => In => Exception
Using undefined instruction => In => Exception
Hardware malfunctions => Either => Either

Two types of exceptions that implementation can generate
1) execution of an undefined instruction
2) arithmetic overflow

EX OF ARITHMETIC OVERFLOW: add $1, $2, $1
save the address of the offending instruction in the Exception Program Counter (EPC) (32 bits) then transfer control to the op. sys. at some specified address. The op sys takes over after that, uses EPC to determine where to restart execution of the program

The op. sys. must know the reason & instr. for exception
MIPS uses a status register (Cause register (32 bits)) that holds the reason for exception.
Second method:
Vectored interrupt = an interrupt for which the address to which control is transferred is determined by the cause of the exception
Ex: undefined instruc = 8000 0000 hex
arithmetic overflow = 8000 0180 hex

Exceptions are another form of control hazard in pipelined implementation
Just as we did for branches, we must flush the instr. that follow the "add" instr. from the pipeline and begin fetching instrs from the new address

To flush instr. in the ID stage, we use a multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is "or"ed w/ a stall signal from the hazard detection unit to flush during ID. To flush in other phases, use EX.flush, etc.

To fetch instructions from exception address, add an additional input to the PC multiplexor that sends the exception address to the PC.

Use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage

Save the address of the offending instruction in the EPC. (actually save address + 4)

undefined instruction = ID stage arith = EX stage
invoking the op. sys. = EX stage
Exceptions are collected in the Cause reg. in a pending exception field so that hardwire can interrupt based on later exceptions, once the earliest has been serviced

Imprecise interrupt/exception = interrupts/exceptions in pipelined computers that are not associated w/ the exact instruction that was the cause of the interrupt/exception

precise = is associated w/ correct instr. in pipelined computers

Instruction-level parallelism (ISP) = the parallelism among instructions

2 Methods for increasing potential amount of ISP:

1) increasing depth of the pipeline to overlap more instr.
   e.g. splitting washer into 3 steps → wash, rinse, spin
   moves from 4-stage to 6-stage pipeline
   rebalance remaining steps to be same length
2) replicate the internal components of the computer so it can launch multiple instructions in every pipeline stage = Multiple Issue = a scheme whereby multiple instructions are launched in one clock cycle.
   e.g. replace washer/dryer w/ 3 washers/dryers
   extra work to keep all machines busy/transferring
   allows CPI < 1
   e.g. CPI = 0.25, IPC = 4

static multiple issue: an approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution

dynamic multiple issue = " " " " " " " " " " " " " "
decisions made during execution by the processor

2 Primary responsibilities w/ multiple-issue pipeline:

1) Packaging instructions into issue slots

Issue slots = the positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint

2) Dealing w/ data and control hazards

Static MI:

Issue packet = the set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor

(think of as a single instruction allowing several operations in certain predefined field =

= Very Long Instruction Word (VLIW))

Use latency = # of clock cycles between a lw and an instr. that can use the result of the load w/out stalling the pipeline

loop unrolling

register renaming

antidependence = name dependence

Memory Technologies

4 Main:

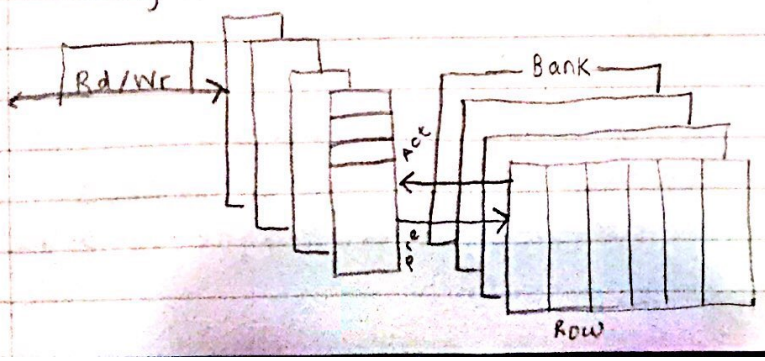
Ex) Main memory (implemented from DRAM)
caches (SRAM)

DRAM less costly than SRAM
substantially slower
significantly less area per bit of memory
larger capacity
Personal Mobile Devices (Flash memory)

SRAM (semiconductor mem) → 0.5-2.5 ns → $500-$1000
DRAM (semiconductor mem) → 50-70 ns → $10-$20
Flash (semiconductor mem) → 5,000-50,000 ns → $0.75-$1.00
Magnetic Disk → 5,000,000 ns - 20,000,000 → $0.05-$0.10

SRAM =
- memory arrays w/ single access port (either read or write)
- fixed access time, though R/W access times differ
- don't need to refresh
- minimal power to retain in standby mode

DRAM =
- value kept in a cell is stored as a charge in a capacitor
- single transistor used to access stored charge (R/W)
- periodically refreshed
- two-level decoding structure, allows to refresh an entire row w/ a read cycle followed immediately by a write



Pre = precharge
opens/closes row
Act = activate sends
row address to buffer

Improve performance:

1) row organization, buffer
2) make chip wider
3) synchronous DRAMs or SDRAMS
   (transfer bits in a burst rather than specifying additional address bits)
   Double Data Rate (DDR) SDRAM
4) Internal Organization (multiple banks)
   Address Interleaving = sending an address to several banks to R/W simultaneously

Dual Inline Memory Modules (DIMMs) - small boards that have many DRAM chips

Memory Rank = each subset of chips in a DIMM

Flash Memory = EEPROM
electrically erasable programmable read-only memory

Wear leveling = a controller spreads the writes by remapping blocks that have been written many times to less trodden blocks
lowers potential performance but it is needed

Disk Memory = platters that rotate
covered w/ magnetic recording material on both sides
like video tape
a movable arm containing an electromagnetic coil called a read-write head is located just above each surface

Each disk surface is divided into tracks
track = one of thousands of concentric circles that makes up the surface of a magnetic disk

Each track is divided into sectors = smallest amount of information that is read/written on a disk

sector # - gap - sector info - gap - next sector #

cylinder = all tracks under the heads at a given point on all surfaces

To access data:

1) seek = position a read/write head over the proper track on a disk
2) Wait for desired sector to rotate under head
   rotational latency = usually assumed to be half the rotation time
3) transfer time = time to transfer a block of bits (dpds on sector size, rotation speed, recording density of track)

The Basics of Caches

cache = represents the level of memory hierarchy between the processor and main memory
also used to refer to any storage managed to take advantage of locality of access

Before a request, the cache contains a collection of recent references $X_1, X_2, \ldots, X_{n-1}$, and the processor requests a word, X_n, that is not in the cache. This results in a miss, and X_n is brought from memory into the cache.

Simplest way to assign a location in the cache is based on the address of the word in memory
= direct-mapped cache = each memory location is mapped to exactly one location in the cache.
ex: 8 block cache uses 3 lowest bits ($2^3 = 8$) of the block address
tag = a field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a required word
(tells if a word in cache corresponds to word request)
we need to know a tag should be ignored for empty cache entries ∴ we add a valid bit to indicate whether an entry contains a valid address

recently referenced words replace less recent.

a referenced address is divided into a
- tag field = used to compare w/ the value of the tag field of the cache
- cache index = used to select the block

For the following situation:
- 32-bit address
- a direct-mapped cache
- cache size is 2^n blocks, so n bits are used for the index
- the block size is 2^m words (2^{m+2} bytes), so m bits are used for the word w/in the block, and two bits are used for the byte part of the address

the size of tag field is $32-(n+m+2)$

total # of bits in direct-mapped cache is

$2^n \times$ (block size + tag size + valid field size)

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for valid field, the # of bits in such a cache is

$2^n \times (2^m \times 32 + (32-n-m-2)+1) = 2^n \times (2^m \times 32 + 31 - n - m)$

Ex: How many total bits are required for a direct-mapped cache w/ 16 KiB of data and 4-word blocks, assuming a 32-bit address?

16 KiB = 2^{12} words = 4096 words } $\therefore 2^{10}$ blocks

4 words = 2^2

Each block has 4×32 bits of data, or 128, plus a tag

tag = 32 - 10 - 2 - 2 bits, plus a valid bit

$\therefore$ total cache size = $2^{10} \times (4 \times 32 + (32-10-2-2)+1) = 147$ Kibibits

Block position is given by: (Block address) modulo (# of blocks in cache)

$$\text{Block address} = \frac{\text{byte address}}{\text{bytes per block}}$$

↑ block size : ↓ miss rate : ↑ miss penalty/cost : rate of ↓ miss rate ↓

to avoid performance loss, the bandwidth of main memory is ↑ through widening and interleaving

a cache miss creates a pipeline stall to wait for memory

1) Send the original PC value (current PC - 4) to memory
2) instruct main memory to Perform a read and wait for memory to complete its access
3) write the cache entry, putting the data from memory in the data portion of the entry, writing upper bits of address into the tag field, and turning the valid bit on
4) Restart the IE step

write-through = a scheme in which (SW) writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data is always consisten between the two

write buffer = a queue that holds data while the data is waiting to be written to memory

OR

write-back = a scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced

Evaluating Cache Performance

CPU time = (CPU execution clock cycles + Memory-stall clock cycles) × Clock cycle time

Memory-stall clock cycles come primarily from cache misses

Memory-stall clock cycles = (Read-stall cycles + Write-stall cycles)

Read-stall cycles = $\frac{\text{Reads}}{\text{Program}}$ × Read miss rate × read miss penalty

e-thru: Write-stall cycles = $\left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{write miss penalty}\right)$ + write buffer stalls ← usually small, can ignore

Memory-stall clock cycles = $\frac{\text{instructions}}{\text{program}} \times \frac{\text{misses}}{\text{instruction}} \times$ miss penalty

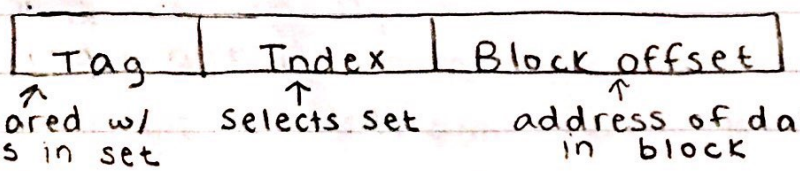
Avg. memory access time (AMAT) = time for a hit + miss rate × miss penalty

Fully associative cache = a block can be placed in any location in the cache
- all entries in cache must be searched

Set associative cache = fixed # of locations (at least 2) where each block can be placed
- n locations for a block is called an n-way set associative cache
- consists of a # of sets, each of which contains n blocks
- Each block in memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of that set

Set position = (Block #) modulo (# of sets in the cache)

Increasing associativity ↓ miss rate : ↑ potential hit time



misses: we choose among blocks in selected set to replace

least recently used (LRU): a replacement scheme in which the block replaced is the one that has been unused for the longest time
(use bits to indicate)

multilevel cache: a memory hierarchy w/ multiple levels of caches, rather than just a cache and main memory

Memory Hierarchy continued

error detection code = a code that enables the detection of an error in data, but not the precise location and, hence, correction of the error

virtual memory = a technique that uses main memory as a "cache" for secondary storage (uses write-back scheme)

physical address = an address in main memory

page = virtual memory block

page fault = an event that occurs when accessed page is not present in main memory

virtual address = an address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed = address mapping

reference bit = set whenever a page is accessed and is used to implement LRU or other replacement schemes

virtual memory allows expansion beyond limits of main memory, and supports sharing of main memory among multiple processes in protected manner.

High cost of page faults

Miss rate ↓ by:

1) enlarging pages / spacial locality
2) mapping between virtual / physical addresses implemented w/ a fully associative page table (TLB)
3) op sys techniques (LRU, reference bit) to choose pages to replace