Handler Can either:

1. Read cause register, transfer to relevant handler.
2. Take corrective action if restartable, use EPC to return.
3. Terminate program and report error using EPC, cause, ...

Exceptions in a pipeline Similar to a mispredicted branch:

1. Complete previous instructions, but flush offending and subsequent instructions.
2. Set cause and EPC values.
3. Transfer control to handler.

Multiple exceptions

1. *Simple method*: flush subsequent instructions: precise exceptions.
2. Complex pipelines may have multiple instructions per cycle, out-of-order completion, etc.

EE2-13 Computer Architecture II

1 Instruction set architecture

Separates software and hardware.

- *Instruction*: vector of bits, must be decoded before executon.

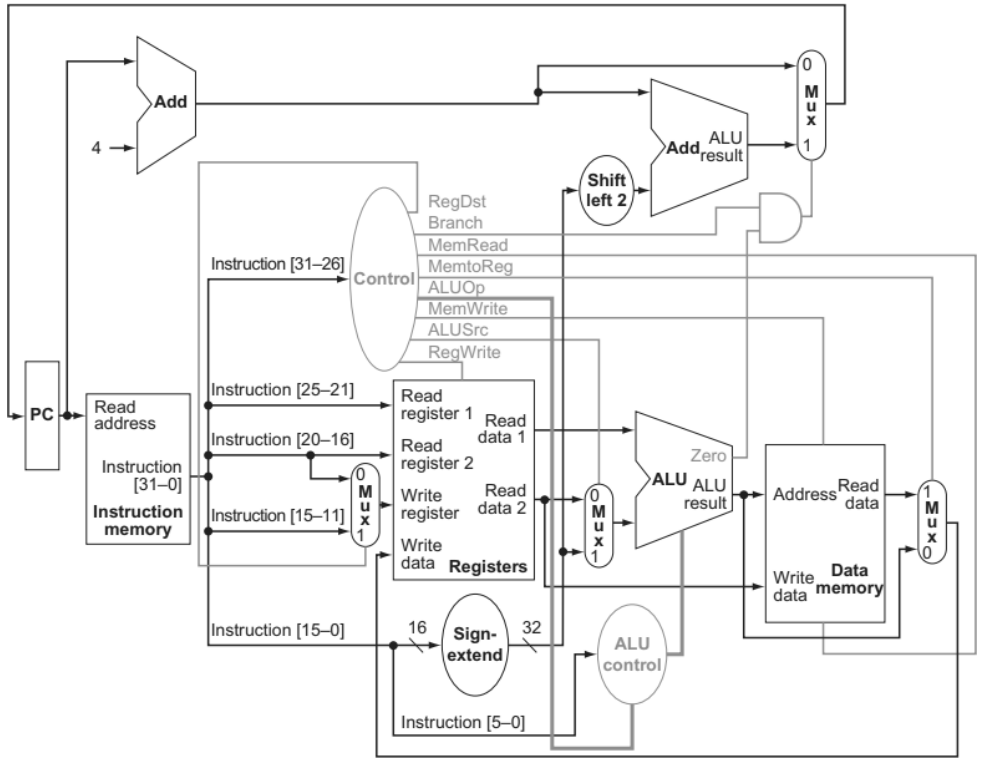
MIPS architecture

- *RISC* (reduced instruction set computer) architecture with *three instruction types*.
- *32 registers*, with $0 wired to 0, all others general purpose.
- *Register-register / load-store* architecture:
  - Most instructions only involve registers. E.g. `add $1, $2, $3`.
  - *Data transfer instructions*. E.g. `lw $8, Astart($19)`.
    - Memory is *big-endian* and access must be *word-aligned*.
- Aim to minimise memory access (may be multi-cycle: slow, non-determinstic).

Representing instructions Three formats.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	`shamt`	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

2 Data and control paths



- *Key idea*: combine datapaths when possible, add control to select.
- A single cycle implementation like this is inefficient: longest path (load instruction) determines clock cycle.

1. *Assume branch not taken*: when a branch is taken, we flush IF, ID and EX stages of pipeline.
2. *Reduce branch delay*: do a register comparison in ID stage (instead of using ALU in EX stage).
   - Bypassed source opereands of a branch could come from ALU/MEM or MEM/WB, so new forwarding logic is required.
   - If a branch directly follows an ALU instruction which produces an operand for the branch, pipeline stall still required.
   - If a branch directly follows a load instruction which produces an operand for the branch, two stall cycles still required.
3. *Dynamic branch prediction*: use runtime information.
   - *Branch prediction buffer*: memory indexed by lower portion 0of address of branch instruction.
     - Marks whether the branch was recently taken or not.
     - Better: 2-bit prediction scheme.
4. *Branch delay slot*: works well for short pipelines with one instruction per clock cycle.

9 Exceptions and Interrupts

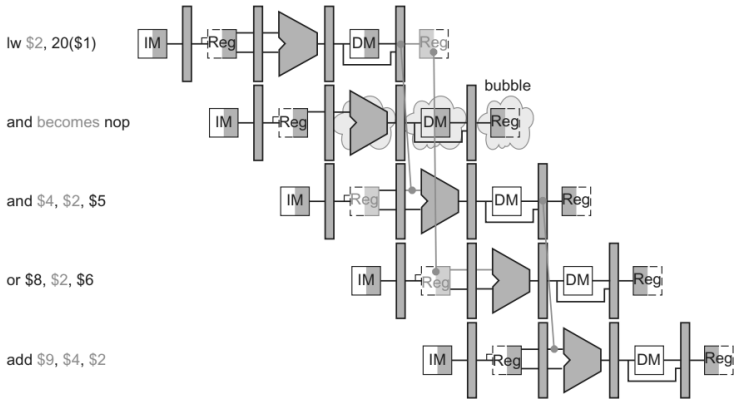
Unexpected events requiring change in flow of control.

- *Exception*: internal signal.
- *Interrupt*: external signal.

MIPS

1. Save the offending instruction in the EPC.
2. Set the cause register.
3. Jump to handler.

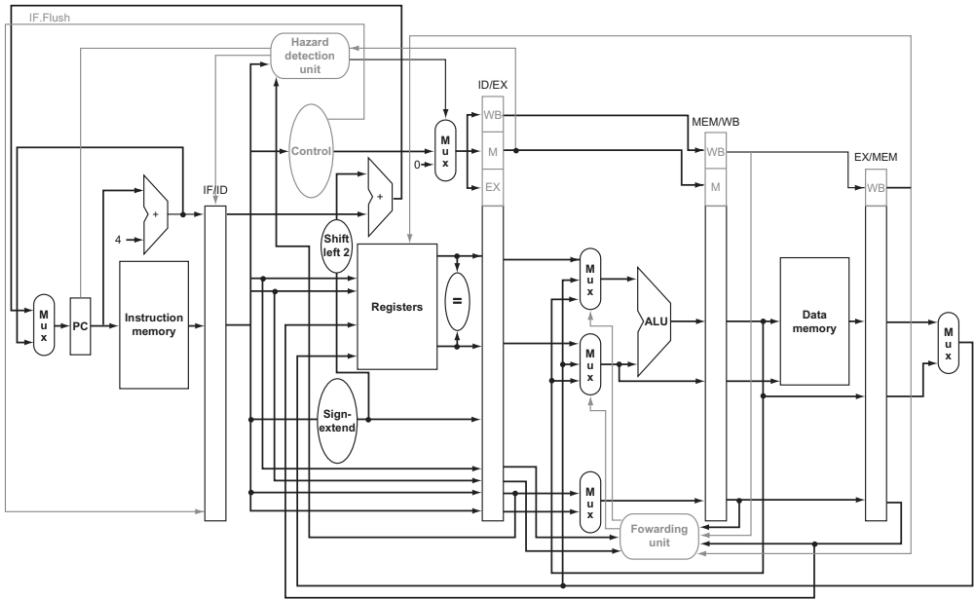
Other architectures Use *vectored interrupts*: handler address determined directly by cause register.



If instruction at ID/EX is load (reads memory), and the load destination register (`Rt`) is a source register in IF/ID, *pipeline stall*:

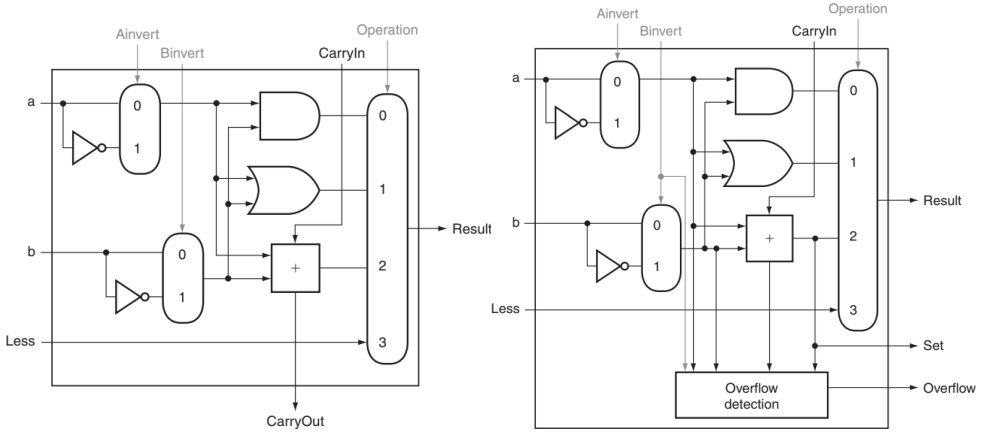
1. Prevent PC register from changing.
2. Prevent IF/ID pipelin register from changing.
3. Set EX, MEM and WB control fields to 0 (no-op).

Control hazards in MIPS

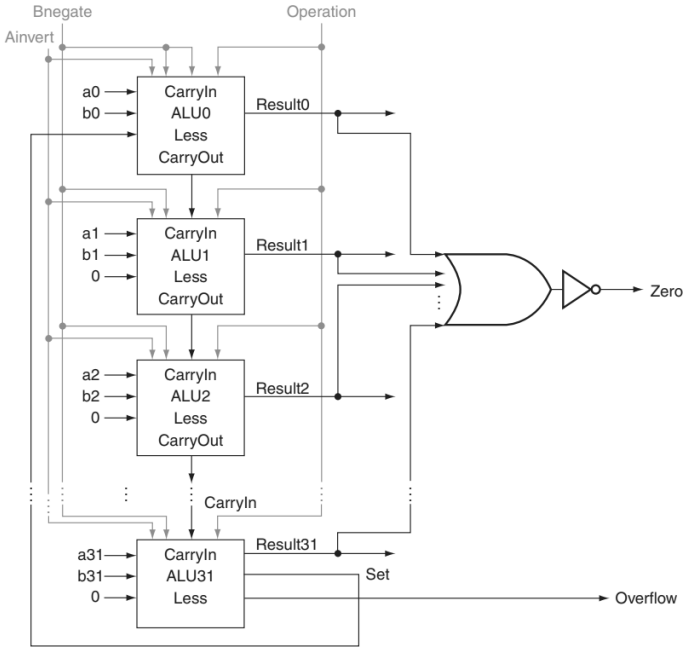


3 Computer arithmetic

ALU Design



- Group components together to form larger repeated unit (less wires crossing, shorter paths).



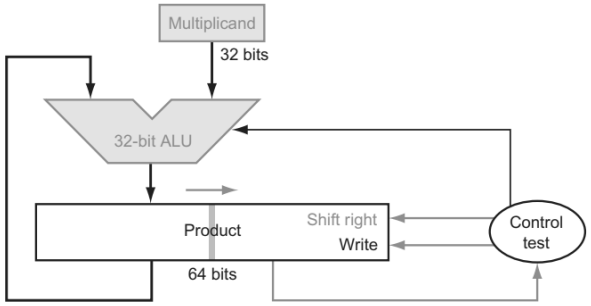
ALU Performance

- Speed limited by propagation delay through slowest combinatorial path:
  - Need to ensure no combinatorial loop.
  - Slowest path usually a carry chain.
  - Max clock rate ≈ 1 / delay of slowest path.

Addition and Subtraction

- Use a full adder.
- *Subtraction*: negate operand by inverting and adding carry of 1.
- *Detecting overflow*: $A, B > 0$ but $A + B \leq 0$ *or* $A, B < 0$ but $A + B \geq 0$.
  - MIPS generates an *exception*. Address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address.
- *Carry-select adder*: for faster addition:
  - Compute for both $c = 0$ and $c = 1$ after n stages.
  - Use multiplexer to choose based on actual c.

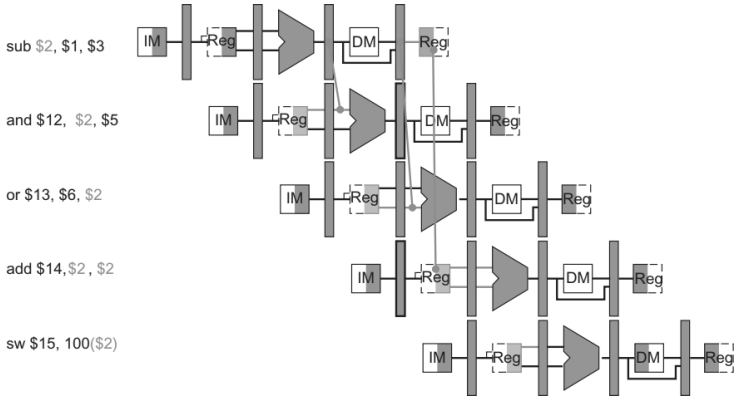
Multiplication



1. Multiplier initally placed in the right half of product reg.
2. If least significant bit is 0, add multiplicand to left half of product reg.
3. Shift product reg right by 1 bit.

2. *Data hazard*: data needed to execute the instruction is not yet available.
   - *Forwarding*: use the result as soon as it is computed (add extra connections in datapath).
   - If the result must be passed backwards, then we must add a *pipeline stall* (bubble).
3. *Control hazard*: instruction that was fetched was not the one that was needed.
   (a) *Pipeline stall*.
   (b) *Branch prediction*.
   (c) *Delayed decision*: have a branch delay slot.

Data hazards in MIPS



1. ID/EX source register = EX/MEM dest. register, given that the instruction at EX/MEM would write, and register is non-zero.
2. ID/EX source register = MEM/WB dest. register, given that the instruction at MEM/WB would write, and register is non-zero (only if no hazard for EX/MEM).
3. Forwarding between WB stage and ID/EX stage can happen directly through the register file.

Load-use data hazard in MIPS

8 Pipelining

Split hardware into stages, biggest delay determines clock period.

1. Need to store partial results and control for future stages.
2. Improves instruction throughput by increasing parallelism.
3. Increases latency for each instruction.

ISA design for pipelining

1. *Instructions same length*: can easily fetch in one cycle and decode in the next.
2. *Few instruction formats*: registers in the same place, so we can read the register file at the same time as working out the instr. type.
3. *Memory operands only in load/store*: we can use execute stage to calculate address for memory access.
4. *Aligned memory access*: only ever need one memory access, so this can be done in a single pipeline stage.
5. *Single result*: writes at most one result, making forwarding simpler.

MIPS pipeline

1. *IF*: instruction fetched from memory.
2. *ID*: instruction deoded and registers read.
3. *EX*: excute operation or calculate address.
4. *MEM*: access memory operand.
5. *WB*: write result back to register.

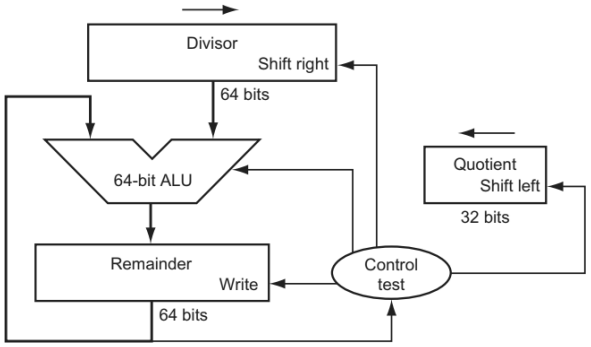
Hazards

1. *Structural hazard*: combination of instructions not supported by hardware (required resource busy).
    - E.g. MIPS pipeline with a single memory cannot do fetch and load/store at the same time.

Booth's Algorithm

- Replace summing $m + 2m + \cdots + 2^{k-1}m$ with $2^k m - m$.
- Replaces k additions with 1 subtraction and 1 addition (and k shifts).
- E.g. $001\,111_2 \times 101_2 = 2^4 \times 101_2 - 101_2$.

Division



1. Subtract divisor from remainder.
2. Shift the quotient to left. Set the least significant bit of the quotient to 1 if remainder was positive or 0 if it was negative.
3. If the remainder is negative, add the divisor back to it.
4. Shift the divisor right.

- *Signed division*: remove signs and add them back.

4 ISA design approaches

1. *CISCs*: complex instruction set computers.
    - Powerful instruction set with variable format.
    - + Dense code.
    - + Only needs simple compiler.
2. *RISCs*: reduced instruction set computers.
    - Simple instructions with fixed format.

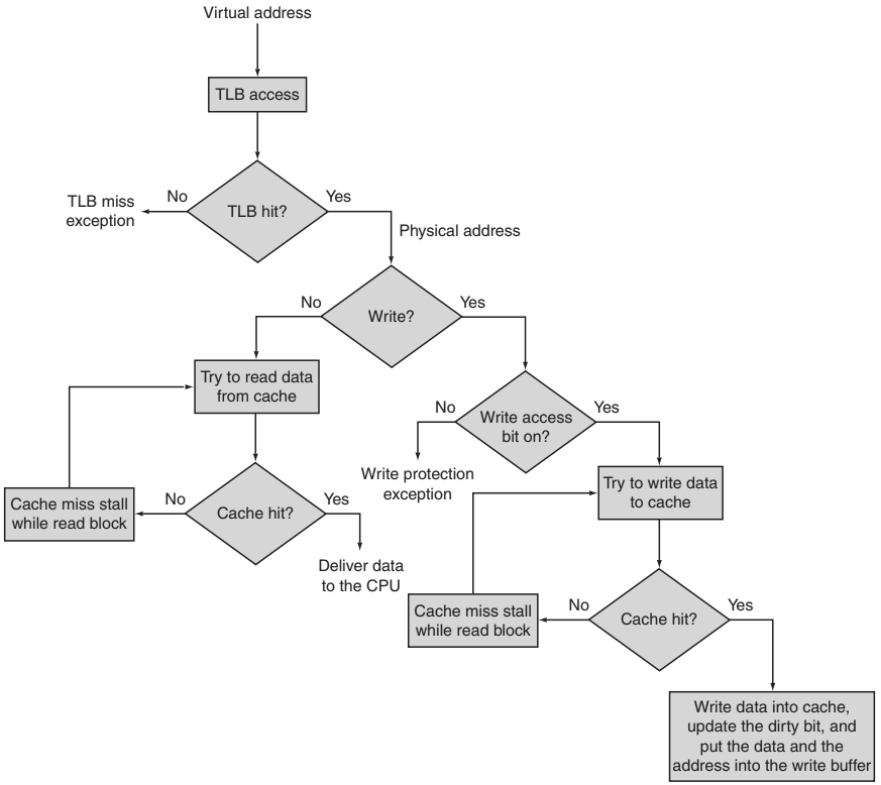
- − Increased instruction count.
- − Increased code size.
- − Requires optimising compiler.
- + Reduced CPI and cycle time.
- + Simple implementation:
  - Adapts well to new technology.
  - Fast/cheap development.
  - Greater confidence in hardware correctness.
  - Smaller chip size/cheaper to manufacture.

Architectures

1. *Stack*. Operands specified implicitly at top of stack.
   - + Dense code.
   - − Less flexible: no random access.
   - − Slow if stack in memory.
2. *Accumulator*. One operand in accumulator register.
   - + Faster than stack architecture.
   - + Only one register (cheap).
   - + Instructions still short.
   - − Frequent memory access still slow.
3. *Register*. Explicit register operands.
   - + Faster than memory and we reduce memory traffic.
   - + Compiler friendly.
   - − High instruction count.
   - − Instructions long (must name all operands).

5 Performance

- *CPI*: clock cycles per instruction
  - Varies based on program and ISA implementation.
- *Clock period*: 1 / clock rate.

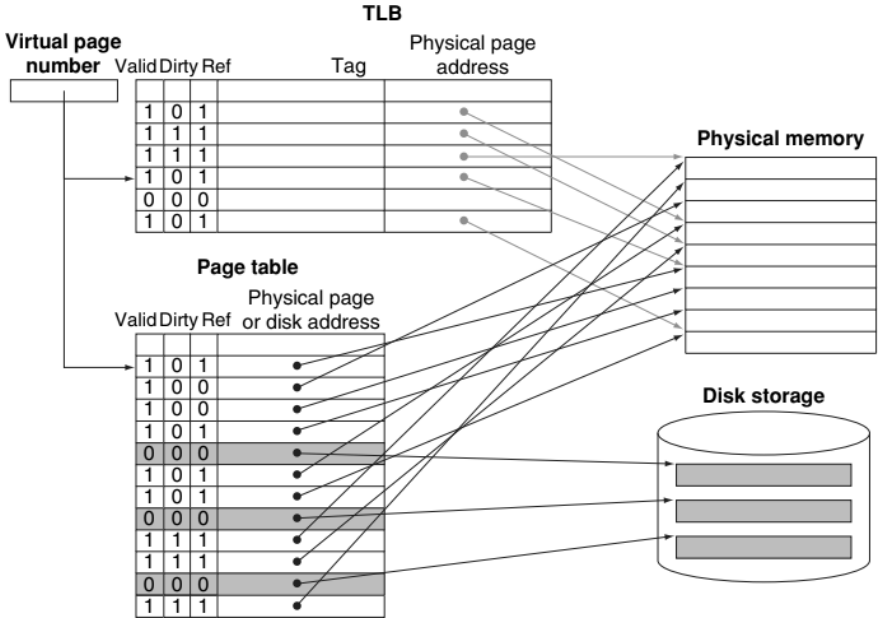


TLB misses Check valid bit of page table entry:

- If 1: copy physical page address to TLB, try again.
- If 0, throw exception, OS handles page fault.
  1. *Cause register* indicates page fault, *EPC* stores address.
  2. CPU switched to supervisor mode, disables further exceptions.
  3. Save state of current process.
  4. Find virtual address causing page fault (depends on instruction/data memory).
  5. Look up page table entry to find disk location.
  6. Choose a physical page to replace (if dirty, must be flushed to disk first).
  7. Start a read to bring referenced page into the chosen physical page.
  8. Run another process during data transfer.

2. Place it somewhere in memory.

Solution: find virtual page number in TLB:



- *TLB hit*: returns physical page addr.
- *TLB miss*: load part of page table into TLB and retry.

TLBs and caches Best scenario: physical address found in TLB is present in cache:

- *Num cycles for P:* num instructions for P × CPI.
- *Execution time for P*: clock period × num cycles for P.

$$\text{Exec. time} = \text{Instr. count} \times \text{CPI} \times \text{Cycle time}$$

- Check dimensions!

MIPS: millions of instructions per second.

- Doesn't take instruction count into account!

Power

$$\text{Power} \propto \text{Capcitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

- *Power wall*: cannot reduce voltage, cannot remove more heat.

Improving performance Beyond the power wall:

1. *Fast, local store* (caches, better RAM).
2. *Concurrent execution* (superscalar, pipelining, multi-threading).
3. *Domain-specific, direct hardware implementations* (FPGAs, no fetch/decode).
4. *Multiprocessors* (e.g. GPUs).
5. *Asynchronous designs* (no global clock).

Evaluating performance

1. Actual target workload.
   - + Representative.
   - − Specific/non-portable.
   - − Difficult to measure.
   - − Hard to idenitify.
2. Full application benchmarks.
   - + Portable.
   - + Widespread.

- + Useful for improvement.
- − Less representative.

3. Small kernal benchmarks.
   - + Easy to use.
   - + Idenitify peak capability.
   - − Typically far away from typical performance.

6 Memory hierarchy

Locality principles

- *Temporal locality*: recently used items tend to be used again soon.
  - Guides cache replacement policy (what to replace when full).
- *Spatial locality*: items close to recently used items tend to be used next.
  - Fetch multiple pieces of data in one transaction.

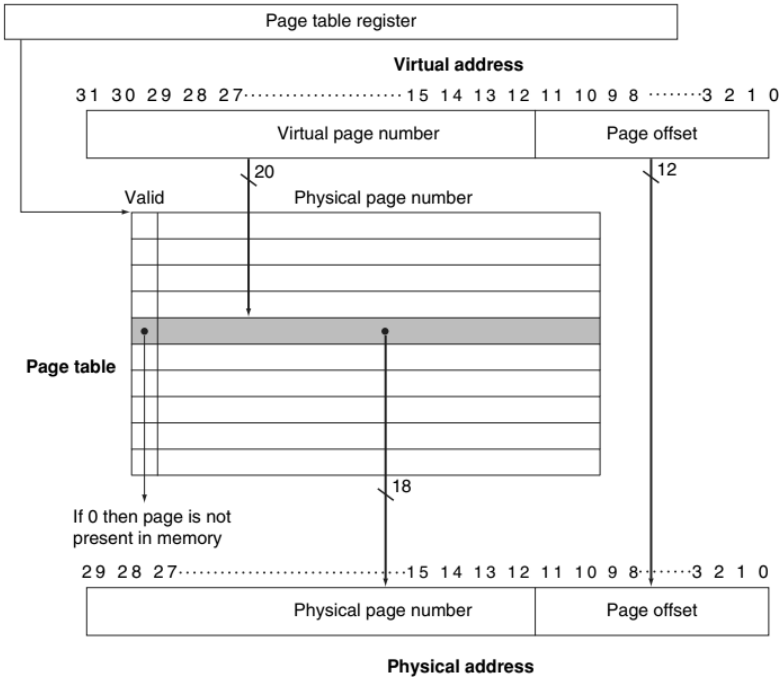
Cache operation Cache organised in *blocks*.

- *Cache hit*: CPU finds required block in cache.
- *Cache miss*: get data from miain memory.
- *Hit rate*: ratio of cache access to total memory access.
- *Hit time*: time to access cache + determine hit/miss.
- *Miss penalty*: time to replace item in cache + transfer to CPU.

Common framework

1. *Plan* a block: can go in one / some / any place.
2. *Find* a block: by index / limited search / full search.
3. *Replace* a block: random / LRU.
4. If *writing* to a block: write through / write back.

Address translation *Virtual memory addresses* are translated to *physical addresses*:



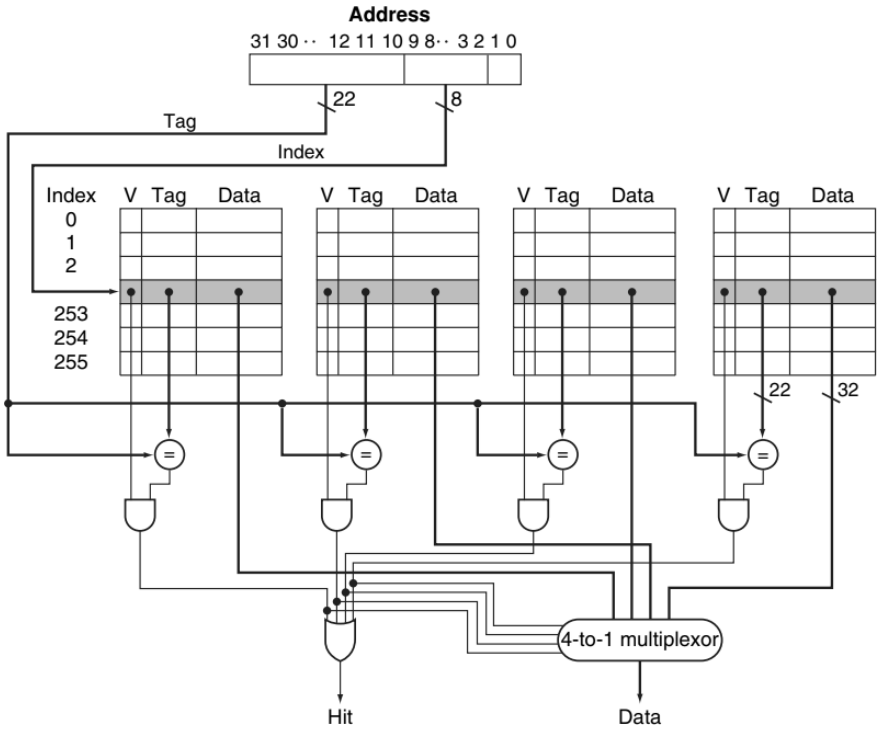
- *Page*: fixed-size block of virtual memory.
- *Page fault*: virtual memory miss.
- *Relocation*: virtual address can be mapped to any physical addr.

Page faults Take millions of cycles to process:

1. Pages shoul be large enough to amortize high access times.
2. Flexible page placement to reduce page faults (e.g. using fully associative schemes).
3. Page faults handled in software, and quite costly algorithms can be used.
4. Write back is the only viable choice for write policy.

Translation Look-aside Buffer On page fault, exception causes OS to make two memory accesses:

1. Use page table to locate required page in disk.



Block replacement policy

- *LRU*: replace least recently used block.
- As cache size increaases:
  - Miss rate decreases.
  - Advantage of LRU over random decrases.

7 Virtual memory

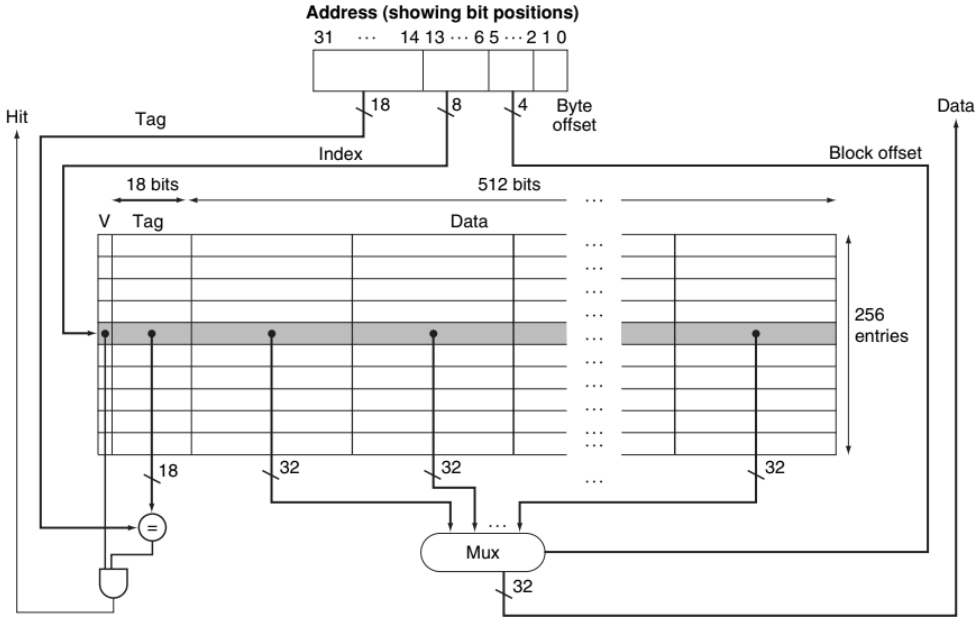
Use main memory as a "cache" for secondary memory.

1. Give each process the illusion of having its own memory.
2. Memory may be shared among multiple programs.
3. Can run programs too large for physical memeory.

Direct mapped cache Each memory locn. mapped to one cache locn. (E.g. using $\mod$).

For a cache with 2^n blocks, each of size 2^m words:

- $32 - (n + m + 2)$ bits are used for the tag field.
- n bits are used for the index.
- m bits are used for the word within the block.
- 2 bits are used for the byte offset.



- Larger blocks:
  - Generally lower miss rate by exploiting spatial locality.
  - Increases transfer time (miss penalty).
  - If cache is small, too few blocks causes early eviction.

Cache write policy Needs to maintian cache coherency.

1. *Write through*: write to both cache and memory.
   - Memory write bandwidth can cause bottleneck.

2. *Write back*: write to cache only.
    - Complex control, requires a dirty bit.
    - Dirty entries must be flushed on eviction.

Cache misses

- *Compulsory misses* (first access).
- *Capacity misses* (insufficient space).
- *Conflict misses* (rigid placement).

Reducing misses:

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

Handling read misses:

1. Send the original PC value (PC - 4) to memory.
2. Instruct main memory to perform a read.
3. Write the cache entry.
4. Restart the instruction at the first step.

Handling write misses (write through): use a write buffer.

- Use a buffer to store data waiting to be written to memory.
- Memory ontroller writes buffer contents to memory.
- Stall if the buffer is saturated.
    - Occurs if CPU cycle time short compared to memory speed, or
    - Possible solution: install a second-level (L2) cache.

Cache performance

$$\text{CPU time} = (\text{Execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}$$

$$\text{Total stall cycles} = \text{Read stall cycles} + \text{Write stall cycles}$$

$$\text{Read stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty (cycles)}$$

$$\text{Write stall cycles} = \frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty (cycles)} + \text{Write buffer stalls (when full)}$$

$$\begin{aligned}\text{Stall cycles} = {} & \text{Instruction count} \\ & \times (\text{Instruction miss rate} + \text{Load/store frequency} \times \text{Data miss rate}) \\ & \times \text{Miss penalty}\end{aligned}$$

$$\text{CPU cycles without stall} = \text{Instruction count} \times \text{CPI (no stall)}$$

$$\text{CPU cycles with stall} = \text{CPU cycles without stall} + \text{Stall cycles}$$

$$\%\text{ of time stalled} = \frac{\text{Stall cycles}}{\text{CPU cycles with stall}}$$

- *Lower CPI*: stall cycles have a greater impact.
- *Lower clock time*: increased miss penalty (in terms of cycles).

$$\begin{aligned}\text{AMAT}_{\text{CPU}} = {} & \text{Hit time}_{\text{L1 cache}} \\ & + \text{Miss Rate}_{\text{L1 cache}} \times (\text{Hit time}_{\text{L2 cache}} + \text{Miss Rate}_{\text{L2 cache}} \times \text{AMAT}_{\text{Mem}})\end{aligned}$$

Associative caches

- *Direct-mapped*: simple, fast access, high miss rates.
- *Fully associative*: costly, slower, low miss rates.
- *n-way set associative*: each address maps to n blocks.