Lecture 3
ISA

Fall 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Falsafi, Hill, Hoe, Lipasti, Shen, Smith, Sohi, and Vijaykumar of Carnegie Mellon University, EPFL, Purdue University, and University of Wisconsin.

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

This Lecture
- Basic caches (1)

Next Lecture:
- Basic caches (2)

Instruction Set Architecture

"Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"

IBM introducing 360 in 1964

- IBM 360 is a family of binary-compatible machines with distinct microarchitectures and technologies, ranging from Model 30 *(8-bit datapath, up to 64KB memory)* to Model 70 *(64-bit datapath, 512KB memory)* and later Model 360/91 *(the Tomasulo).*
- IBM 360 replaced 4 concurrent, but incompatible lines of IBM architectures developed over the previous 10 years

ISA: a contract between hw and sw

- Programmer visible states

  program counter, GPR, memory, status/cntrl
- Programmer visible behaviors (state transitions)

  what to perform? where are the operands? what to perform next?

```
if imem[pc]=="add rd, rs, rt"
then
        pc ⇐ pc+1
        gpr[rd]=gpr[rs]+grp[rt]
```

- *A binary encoding*

  Expected lifetime ~25 years (because of SW cost)

Typical Instructions (Opcodes)

Type	Example Instruction
Arithmetic and logical	and, add
Data transfer	move, load
Control	branch, jump, call, return
System	trap, rett
Floating point	add, mul, div, sqrt
Decimal	addd, convert
String	move, compare

What operations are necessary? *{sub, ld & st, conditional br.}*

Too little or too simple → not expressive enough
- difficult to program (by hand)
- programs tend to be bigger

Too much or too complex → most of it won't be used
- too much "baggage" for implementation.
- difficult choices during compiler optimization

ALU Instructions

ALU instructions combine operands

Number of explicit operands
- two, $r_i = r_i \text{ op } r_j$
- three, $r_k = r_i \text{ op } r_j$

What about zero, one and four?

Orthogonality of operands: registers or memory (+ addressing modes)
- any combo, orthogonal, e.g., VAX
- at least one register, not orthogonal, e.g., IBM 360/370
- all registers, orthogonal but loads/stores, e.g., Cray, RISCs

Register Operands

Why registers?
- faster access
- shorter/simpler address

Accumulator: *a legacy from the "adding" machine days*
- ✓ less hardware
- ✗ high memory traffic
- ✗ likely bottleneck

Stack: *an intuitive programming construct*
- ✓ code density
- ✗ bottleneck while pipelining (why?)
- e.g., Burroughs's Stack Machine for ALGOL, x86 Floating-point, JAVA VM

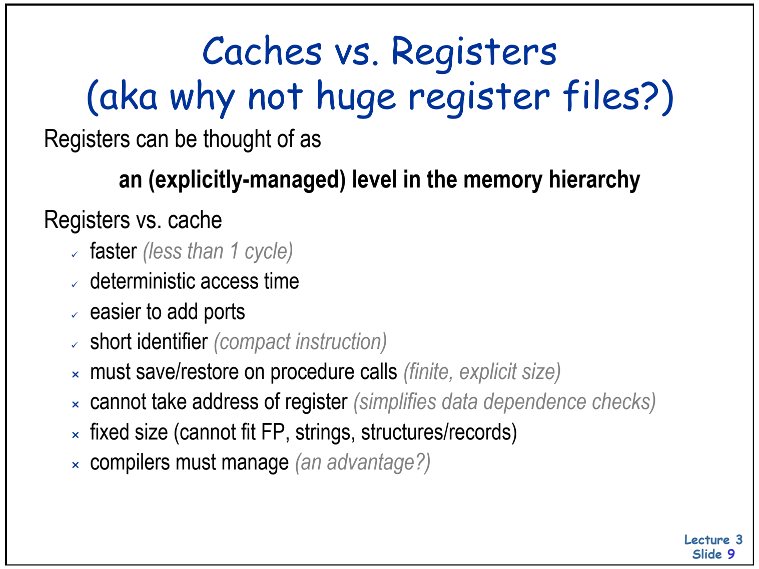
Operand Storage

General Purpose Registers (8 to 256 words):
- ✓ most flexible
- ✗ register must be named
- ✗ code density

An evolving design point:

accumulator → multiple accumulator/scratch registers → accumulator + address registers → arithmetic operations directly on address registers → a small number of general purpose registers (GPR) → RISC 32 GPRs → Itanium's 128 GPRs

What were the driving forces?



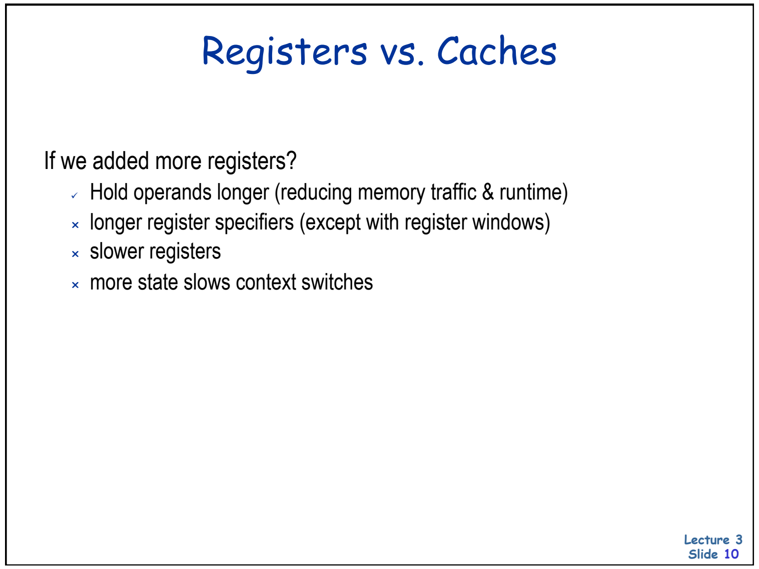
Caches vs. Registers (aka why not huge register files?)

Registers can be thought of as

an (explicitly-managed) level in the memory hierarchy

Registers vs. cache

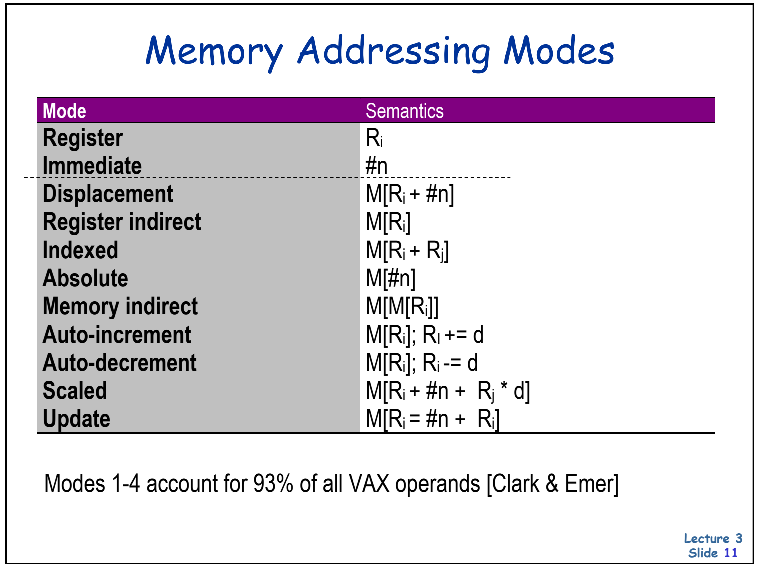
- ✓ faster *(less than 1 cycle)*
- ✓ deterministic access time
- ✓ easier to add ports
- ✓ short identifier *(compact instruction)*
- × must save/restore on procedure calls *(finite, explicit size)*
- × cannot take address of register *(simplifies data dependence checks)*
- × fixed size (cannot fit FP, strings, structures/records)
- × compilers must manage *(an advantage?)*



Registers vs. Caches

If we added more registers?

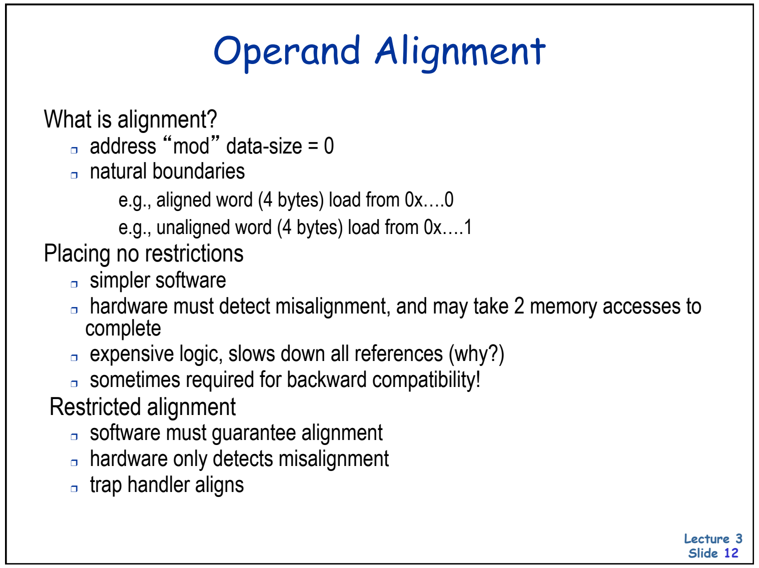
- ✓ Hold operands longer (reducing memory traffic & runtime)
- × longer register specifiers (except with register windows)
- × slower registers
- × more state slows context switches



Memory Addressing Modes

Mode	Semantics
Register	R_i
Immediate	#n
Displacement	$M[R_i + \#n]$
Register indirect	$M[R_i]$
Indexed	$M[R_i + R_j]$
Absolute	$M[\#n]$
Memory indirect	$M[M[R_i]]$
Auto-increment	$M[R_i]; R_i += d$
Auto-decrement	$M[R_i]; R_i -= d$
Scaled	$M[R_i + \#n + R_j * d]$
Update	$M[R_i = \#n + R_j]$

Modes 1-4 account for 93% of all VAX operands [Clark & Emer]



Operand Alignment

What is alignment?

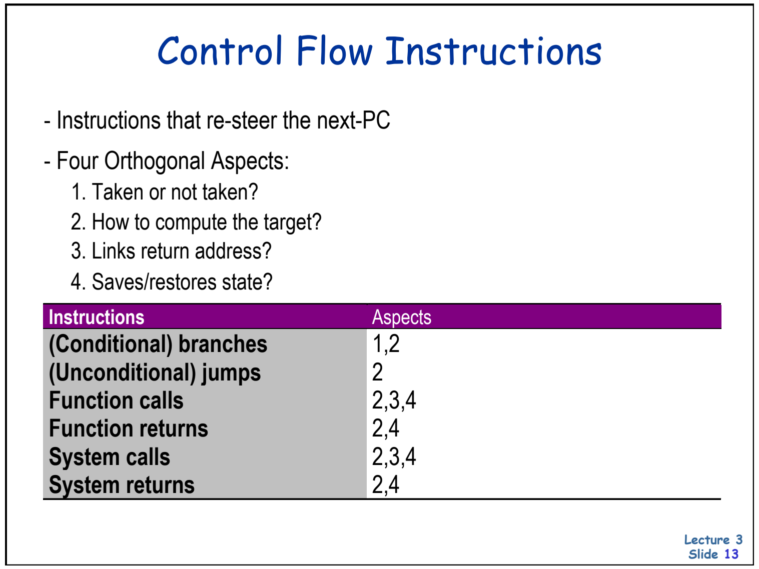
- ▫ address "mod" data-size = 0
- ▫ natural boundaries
  - e.g., aligned word (4 bytes) load from 0x….0
  - e.g., unaligned word (4 bytes) load from 0x….1

Placing no restrictions

- ▫ simpler software
- ▫ hardware must detect misalignment, and may take 2 memory accesses to complete
- ▫ expensive logic, slows down all references (why?)
- ▫ sometimes required for backward compatibility!

Restricted alignment

- ▫ software must guarantee alignment
- ▫ hardware only detects misalignment
- ▫ trap handler aligns



Control Flow Instructions

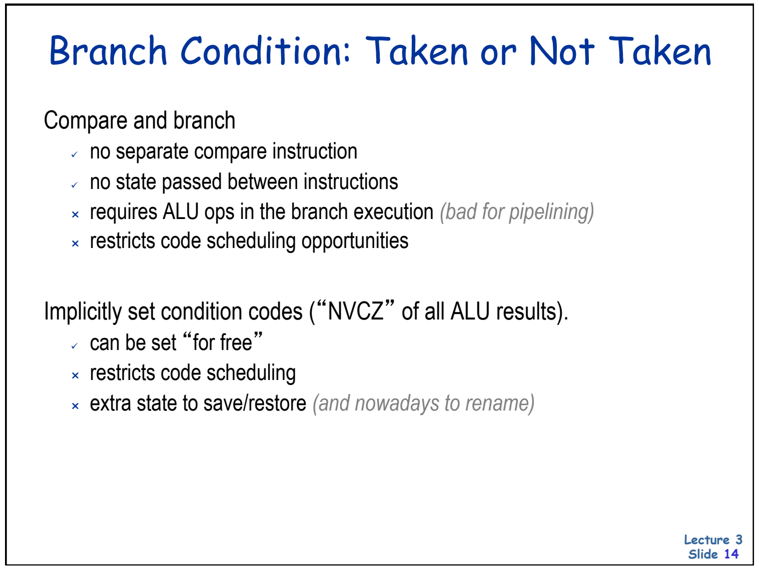
- Instructions that re-steer the next-PC

- Four Orthogonal Aspects:

1. Taken or not taken?
2. How to compute the target?
3. Links return address?
4. Saves/restores state?

Instructions	Aspects
(Conditional) branches	1,2
(Unconditional) jumps	2
Function calls	2,3,4
Function returns	2,4
System calls	2,3,4
System returns	2,4

Lecture 3
Slide 13



Branch Condition: Taken or Not Taken

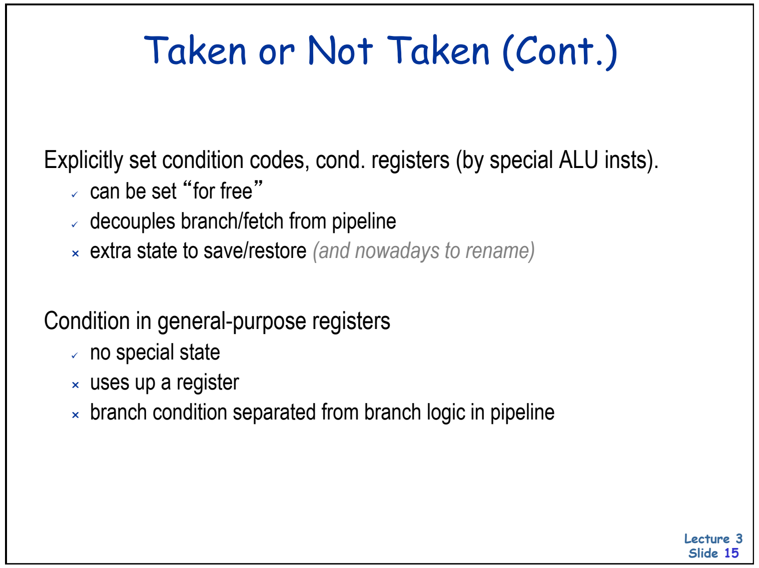
Compare and branch

- ✓ no separate compare instruction
- ✓ no state passed between instructions
- × requires ALU ops in the branch execution *(bad for pipelining)*
- × restricts code scheduling opportunities

Implicitly set condition codes ("NVCZ" of all ALU results).

- ✓ can be set "for free"
- × restricts code scheduling
- × extra state to save/restore *(and nowadays to rename)*

Lecture 3
Slide 14



Taken or Not Taken (Cont.)

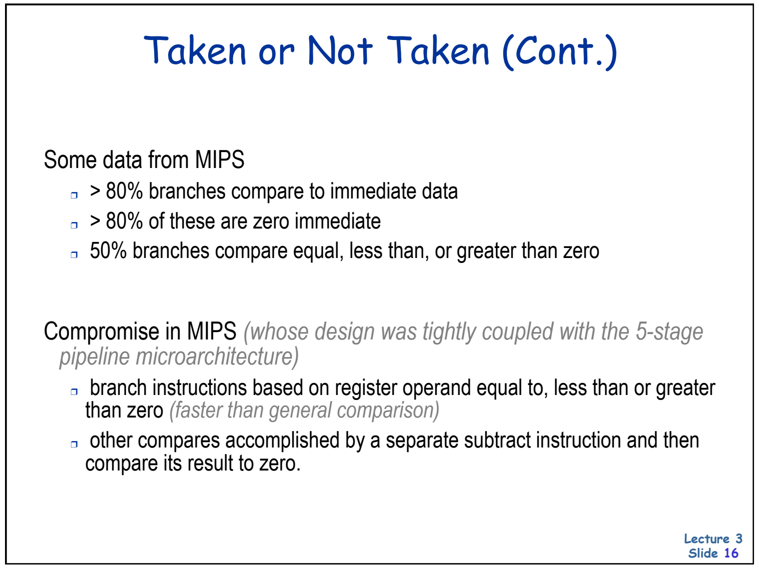
Explicitly set condition codes, cond. registers (by special ALU insts).

- ✓ can be set "for free"
- ✓ decouples branch/fetch from pipeline
- × extra state to save/restore *(and nowadays to rename)*

Condition in general-purpose registers

- ✓ no special state
- × uses up a register
- × branch condition separated from branch logic in pipeline

Lecture 3
Slide 15



Taken or Not Taken (Cont.)

Some data from MIPS

- ❑ > 80% branches compare to immediate data
- ❑ > 80% of these are zero immediate
- ❑ 50% branches compare equal, less than, or greater than zero

Compromise in MIPS *(whose design was tightly coupled with the 5-stage pipeline microarchitecture)*

- ❑ branch instructions based on register operand equal to, less than or greater than zero *(faster than general comparison)*
- ❑ other compares accomplished by a separate subtract instruction and then compare its result to zero.

Lecture 3
Slide 16

Where is the Target?

Arbitrary specifier
- ✓ orthogonal
- ✗ more bits to specify => more time to decode
- ✗ branch execution and target separated in pipeline

PC-relative with immediate (via simple addition or masking)
- ✓ short immediate sufficient — #bits < 4 (47%), < 8 (94%)
  - *the immediate is combined with the PC by addition or masking; hence can't jump too far from current PC value*
  - *MIPS: conditional ±2^{17}byte, unconditional ±2^{29}byte*
- ✓ position independent (relocateable code)
- ✓ target computable in the branch unit
- ✗ target must be known statically
  - → other techniques needed for function returns, virtual functions, distant jumps

Where is the Target? (Cont.)

Register
- ✓ short specifier
- ✓ can jump anywhere
- ✓ can have dynamic target *(bad for branch prediction)*
- ✗ branch and target are separated in the pipeline

Vectored traps (for system calls)
- ✓ protection
- ✗ implementation headaches

Where is the Target? (Cont.)

Common compromises

Type	Where is the target?
(Conditional) branches	PC-relative
(Unconditional) jumps	PC-relative and register
Function calls	PC-relative and register
Function returns	register
System calls	trap
System returns	register

Calling Conventions: Saves/Restores State?

What state?
- function calls ==> PC (linking), registers
- system calls ==> registers, flags, PC, PSW, etc.

Software register save
- caller saves "caller-saved" registers in use
- callee saves "callee-saved" registers that it will use

Hardware register save
- IBM STM, VAX CALLS
- faster?

Examples
- RISC generally does no hardware register saving
- SPARC has implicit saving with register windows
- Itanium has general hardware save and restore

64-bit MIPS Instruction Set

Decedent from Stanford MIPS Project *(Microprocessor without Interlocked Pipeline Stages)*

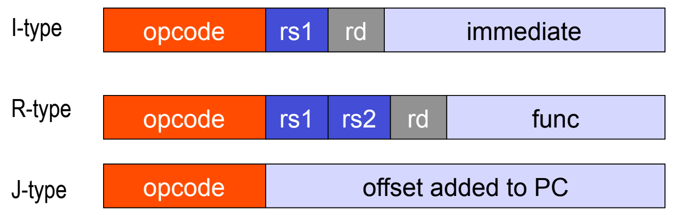
- 8-, 16-, 32-, 64-bit integers
- 64-bit floating point
- load/store architecture
- register, displacement, immediate, and register indirect
- 32-bit instructions
- 3 fixed length formats
  - 32 64-bit GPRs (R0 = 0)
  - 32 64-bit FPRs
  - FP status register
  - no CCs

Designed for compiler generated code

- efficient hardware implementations

- aggressive optimizations

MIPS Instruction Formats



I: ALU register-immediate, load/store, branches, jump register

R: RRR ALU ops

J: unconditional jumps

MIPS (Cont.)

Data transfer

- load/store byte, halfword, word, double-word
- load/store sign-extension
- load/store FP single/double
- moves between GPRs and FPRs
- Little Endian/Big Endian software selectable

ALU

- add/sub
- mul/div
- and, or, xor
- shifts: ll, rl, ra
- sets

MIPS (Cont.)

Control

- branches == 0, <> 0
- conditional branch testing FP bit
- jump, jump register
- jump&link, jump&link register
- trap, return from trap

FP

- add/sub/mul/div single/double
- fp converts, fp set

Memory Systems

Basic caches
- introduction
- fundamental questions
- cache size, block size, associativity

today

Advanced caches

Main memory

Virtual memory

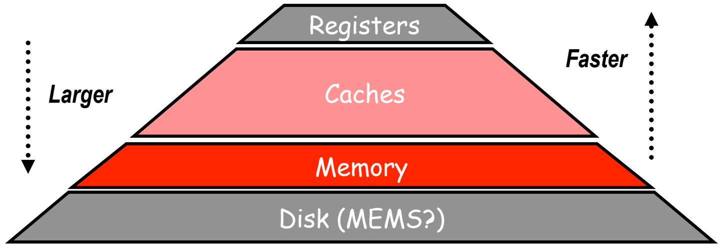
Motivation

CPU can only go as fast as memory!
- memory reference/inst x bytes-per-reference x IPC/cycle time
- In 1990: (1+0.2) x 4 x 1 / 2ns = 2.4 GB/s
- In 2000: (1x4+0.2x8) x 3 / 0.3ns = 56 GB/s
-

Want storage memory:
- as fast as CPU
- as large as required by all of the running applications

Memory Hierarchy

Make common case fast:
- common: temporal & spatial locality
- fast: smaller more expensive memory



Storage Hierarchies

Storages are layered by hierarchies away from the CPU in the order of
- increasing latency (t_i) $t_i < t_{i+1}$
- increasing size (s_i)
  - $\Rightarrow$ decrease unit cost (c_i) $s_i < s_{i+1}$, $c_i < c_{i+1}$
- decreasing bandwidth (b_i) $b_i > b_{i+1}$
- increasing xfer unit (x_i) $x_i < x_{i+1}$

Level 0 Registers — ISA feature

Memory Abstractions

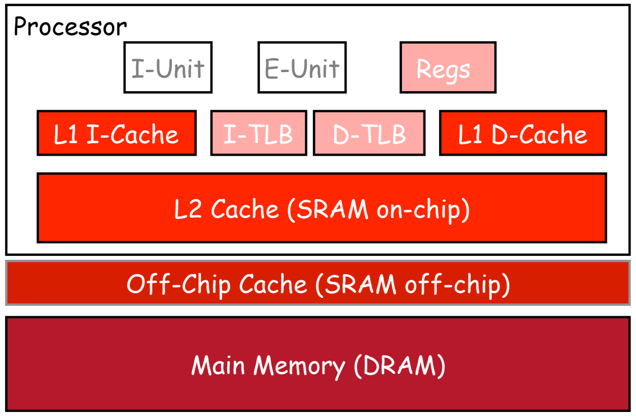
Level 1 (n levels of) Caches

Level 2 Main Memory (Primary Storage)

Level 3 Disks (Secondary Storage)

Level 4 Tape Backup (Tertiary Storage)

Processor/Memory Boundaries



Caches

An automatically managed hierarchy

"A hiding place, esp. of goods, treasure, etc." -- OED

Keep recently accessed block

- temporal locality

Break memory into blocks (several bytes)

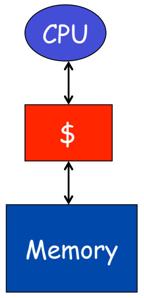
and transfer data to/from cache in blocks

- spatial locality

A lot of architectures opt for software

managed scratch-pad memory instead

e.g. Cray-1, embedded processors, Why??



Cache Performance

Assume

- Cache access time is equal to 1 cycle
- Cache miss ratio is 0.01
- Cache miss penalty is 20 cycles

Mean access time

= Cache access time + miss ratio * miss penalty

= 1 + 0.01 * 20 = 1.2

Typically

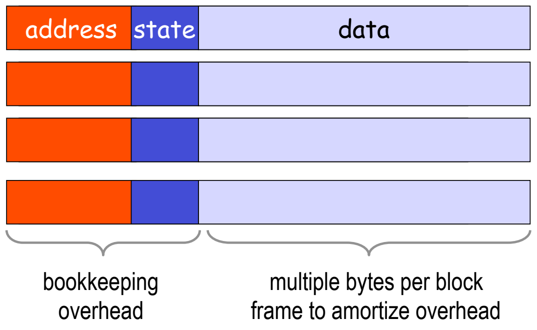
- level-1 is 16K-64K, level-2 is 512K-4M,memory is 8G-2TB
- level-1 as fast as the processor *(increasingly 3-cycles)*
- level-1 is 1/10000 capacity but contains 98% of references

Memoization & amortization

Cache (Abstractly)

Keep recently accessed block in "block frame"

- state (e.g., valid)
- address tag
- data



Cache (Abstractly)

On memory read
- if incoming address corresponds to one of the stored address tag then
  - HIT
  - return data
- else
  - MISS
  - choose & displace a current block in use
  - fetch new (referenced) block from memory into frame
  - return data

- Where and how to look for a block? (Block placement)
- Which block is replaced on a miss? (Block replacement)
- What happens on a write? Write strategy (Later)

Terminology

block (cache line) — minimum unit that may be present

hit — block is found in the cache

miss — block is not found in the cache

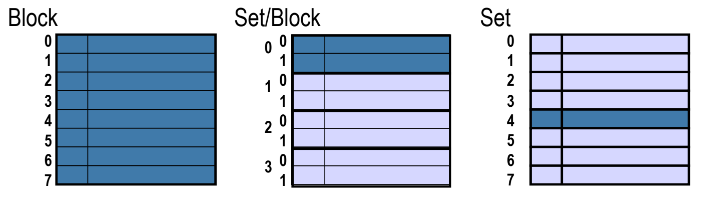
miss ratio — fraction of references that miss

hit time — time to access the cache

miss penalty
- time to replace block in the cache + deliver to upper level
- access time — time to get first word
- transfer time — time for remaining words

Block Placement

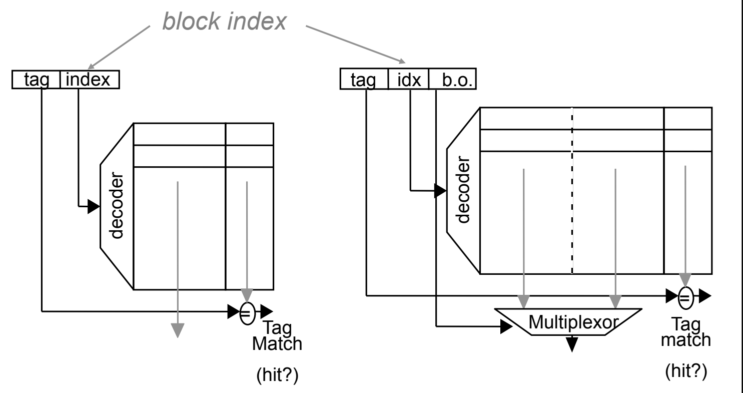
Where does block 12 (b’1100) go?



Fully-associative	Set-associative	Direct-mapped
block goes in any frame	a block goes in any frame in exactly one set	block goes in exactly one frame

(think all frames in 1 set) ⟷ (frames grouped into sets) ⟷ (think 1 frame per set)

Direct Mapped Caches



Don’t forget to check the valid/state bits

Cache Block Size

Each cache block frame or (cache line) has only one tag but can hold multiple “chunks” of data

- reduce tag storage overhead

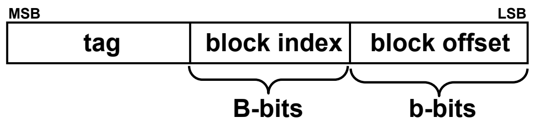
  In 32-bit addressing, an 1-MB direct-mapped cache has 12 bits of tags

  4-byte cache block $\Rightarrow$ 256K blocks $\Rightarrow$ ~384KB of tag
  128-byte cache block $\Rightarrow$ 8K blocks $\Rightarrow$ ~12KB of tag

- the entire cache block is transferred to and from memory all at once

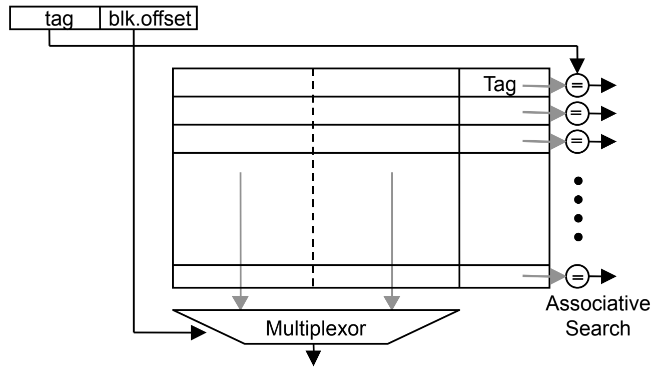
  good for spatial locality because if you access address i, you will probably want i+1 as well (prefetching effect)

Block size = 2^b; Direct Mapped Cache Size = 2^{B+b}



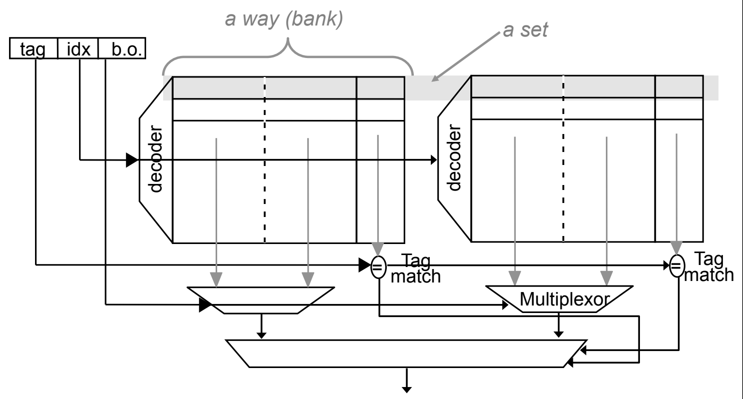
Lecture 3
Slide 37

Fully Associative Cache



Lecture 3
Slide 38

N-Way Set Associative Cache



Cache Size = N x 2^{B+b}

Lecture 3
Slide 39