Computer Architecture

- Aspects → circuits
- Microarchitecture
  - logical organization that implements ISA
  - processor designer
- ISA
  - Instructions
  - programmer / compiler
- Technology
- Requirements
  - User
  - Programming Languages
  - Compilers
  - → Operating System
  - Applications
- Organization
  - old → 冯诺依曼结构 Von Neumann
    - key concepts
      - program and data are stored in same memory, both addressable
      - sequential execution
    - * ALU centric (运算器)
      - data transfer between I/O and memory must passed through ALU
  - nowadays
    - * memory-centric
      - direct data transfer between memory and I/O
      - better CPU performance
- Terms
  - word 机器字长 — no. of bits a CPU can process one time ⟷ Data Bus width and register width
  - 存储字长 memory width
    - *MAR =
    - = ⟷ MDR
  - 指令字长 instruction length

execution process

1) get instruction
- PC —address→ MAR
- DBUS —data→ MDR
- MDR → IR

2) decode instruction
- CU —decode→ IR
- ~~IR —data add→ MAR~~
- ~~DBUS —data→ MDR~~

3) execute (CU sends control signals)
- IR —data add→ MAR
- DBUS —data→ MDR
- MDR —data→ RI

- control unit
  - Instruction Register IR
    - -store instruction
  - Program Counter PC
    - -store next instruction's address
- memory
  - memory address register MAR
    - -store address
  - memory data register MDR
    - -store data

multiple instructions per cycle

thread level parallelism

Graphics Processing Units GPU

Special-purpose logic

2) implicit parallelism

3) Explicit Parallelism

4) Heterogeneous Processing

complex processor designs

dynamic scheduling

1) microprocessor
- Dennard scaling

Revolution

5) Current Trends

parallelism (chip level)

security

power-conscious

specialization

$$\text{speedup} = \frac{P_{after}}{P_{before}} = \frac{T_{before}}{T_{after}}$$

reason: reduce the total latency

Common case first
- give resource to common case

Latency
Execution/response time - overall time

Performance

Performance

Design

Amdall's Law
- theoretical speedup is limited by part of the unchanged tasks (usually ~~sequential tasks~~ serial)

$$\text{theoretical speedup} = \frac{1}{(1 - R_{improved}) + \frac{R_{improved}}{S_{improved}}}$$

in extreme case ∞

$$\text{max speedup} = \frac{1}{1 - R_{improved}} = \frac{1}{R_{remain}}$$

$\frac{1}{x}$

Performance Metric

three aspects are dependent to each other

(CPU 执行时间)
CPU time
- time used in CPU

=

Instructions count (IC) x Average instruction time

=

Cycle Per Instruction CPI x Clock Cycle Time CT

$\frac{1}{x}$

Clock Cycle Frequency

Little's Law

Items = average arrival rate (throughput) x average wait time (latency)

性能 Performance

$\frac{P_{after}}{P_{before}}$ = 加速比 speedup

$= \frac{T_{before}}{T_{after}}$

吞吐量 Throughput/ 带宽 bandwidth
- tasks per wait time

parallelism increase throughput

Floating Point Operation Per Second FLOPS

Million Instruction per Second MIPS

=

Instruction per Cycle IPC x Clock Cycle Frequency

Parallelization

weak scaling
- increase problem size

strong scaling
- reduce serial code

$$\frac{1}{(1 - R_{parallel}) + \frac{R_{parallel}}{Threads}}$$

signed bit-shifting

↓ left-shift
- works for + -

↘ right-shift
$\lfloor (x+2^k-1)/2^k \rfloor$
- divide 2^k
修正负数偏差

R1 最低位+移除数 - change x → +2^
Booth algorithm
补码

01 ⌊ ⌋ 10
RO=RO+R2 ; RO=RO-R2
⌊ ⌋
↓
RO,R1,未移除数 >>1
↓
n次

原码
R1 最低位
1 ⌊ ⌋ 0
RO=RO+R2 ; RO=RO+0
⌊ ⌋
↓
RO,R1 >>1
↓
n次
simple, slow

加法器 ↑

-no sign
Overflow ←

Multiplication ↑

same sign addition
or
different sign subtraction
↑
溢出 Overflow
↑
Addition / Subtraction ↑
↑
加法器/减法器 → use xor gate to determine sign

符号设为00/11, if 01,10 overflow
↑
→ 检测
- → 进位位相异或 C_n xor C_{n-1} (快)
- → 符号位判断(个量)

Logical 逻辑
↑ ! not
→ || or
→ && and

throw away higher bits
* may cause data loss
← Truncating 截断

Type Conversion

Integer → Operation

signed extension
- 前面加符号位 ← 位扩展 Bit Extension

0扩展 Unsigned
- 前面加0

$$[X]_{补} = \begin{cases} X, 0 \le X < 2^n \\ 2^{n+1} + X, -2^n \le X < 0 \end{cases}$$

^: 模 modulo

$[X]_{补} - [Y]_{补}$
=
$[X]_{补} + [-Y]_{补}$
* can be directly calculated

Coding →
- *补码 Two's complement
  - 1) positive: unchanged
  - 2) negative: 取反加1 / last 1 unchanged, reversed rest of front
- 反码 One's complement
  - first bit as sign
  - negative: reversed binary
- 原码 Signed magnitude
  - first bit as sign (0:+ ; 1:-)

has 2 0s

~ one's complent 取反
^ xor
& > ^ > | > && > ||
& ← Bitwise 按位 → | or
and

(divide 2^k)
右移 >>k
right-shift
- 算术
  - arithmetic shift
  - if sign bit = 1
  - add 1 as highest bit
- 逻辑
  - logical shift
  - shift in zeros no matter what

($\times 2^k$)
左移 <<k
left-shift
- logical = arithmetic
- may overflow

Floating point 浮点数

Rounding
- Guard bit
  - first bit to be removed
- Round bit
  - 2^{nd} bit
- sticky bit
  - if any 1 after 3^{rd} bit =1
  - else =0

最近舍入 Round to even
- >half, 进位
- <half, 舍
- =half round to even

decimal→binary
1) 确定符号
2) 转 binary
3) 移浮点 → 规格化数
4) 确定 E和M

范围/精度
if 字长 const, 阶码↑精度↓

Conversion/casting
- double/float → int (truncates)
- double → float (truncates)
- int / float → double (precise)

编码 Coding

S	E	M

- sign 符号位
  - 0: positive
  - 1: negative
- Exponent 阶数
- Mantissa 尾数

$N=(-1)^S \times M \times 2^E$

规格化
- IEEE754 - 浮点前缺省1
- 普通 - 尾数最高位=1

IEE 754 标准

乘法
same as 加法

* 1) 阶码相加
* 2) 规格化: if M≥2, M>>1→e+1

加法
1) 对阶 (小→大)
2) M 相加
3) 规格化 (if first==1)
4) 舍入 M + 规格化
5) check overflow

☆ 符号位不参与运算

Properties
- e uses 原码
- M缺省1位, 范围↑
- 0 = Int(0)
- direct comparison with unsigned int → except → -0=0
  - NaN is the largest
  - different sign

Types
- 扩展精度 (Intel) Extended precision 80-bits
  - S=1 E=15 M=63/64
- single precision 单精度 32-bits
  - S=1 E=8 M=23
  - $(-1)^S \times 1.M \times 2^{e-127}$
  - $1 \le e \le 254$, e=0, M=0
- Double precision 双精度 64-bits
  - S=1 E=11 M=52
  - $(-1)^S \times 1.M \times 2^{e-1023}$

NaN | -规格化数 | -0/+0 | +非规格化 | +规格化 | +∞ | NaN

- NaN: e=255, M=1xxx
- -∞: e=255, M=0
- -非规格化数 subnormal: e=0 f≠0
- +∞: e=255, M=0
- NaN: e=255, M=0xxx

Alignment 数据对齐
- 字对齐
  - 快
  - 接口简单
- 不对齐
  - 慢
  - 接口复杂
  - 空间少

数据存储字节顺序
- Big endian
  - 低字节高地址
  - 左→右
- Little Endian
  - 相反

Memory System Architecture

RAS * Precharge happens here

- tRP
- tRCD
- CAS
- CL
- Data
  - same row as last time → Burst 突发传输 -连续地址连续传输
  - different row → tRP → Output

tRCD

Row to Column Delay

tRP

RAS Precharge 行预充电 (关闭行)

CL

CAS Latency

CAS → Data Delay

答案

同步 DRAM

Synchronous DRAM (SDRAM)

- Double Data Rate SDRAM DPR 2/cycle
- SDRAM ← add clock supports burst
  - burst mode with critical word first
  - organization
    - wide bus -read all words in parallel
      - simple ← ↑ bandwidth
      - expensive
      - conflict
        - 取指令 -程序转移
        - 读操作数 -不确定
        - 写 -需读
        - 读写 -不并行
    - 多体交叉 Interleaved
      - Dived memory to ^ banks
      - ^ ≥ cycle per word
        - hide latency by parallelism
        - ↑ bandwidth $t = T + (\lambda - 1)\tau$ (T: memory cycle; λ: word count; τ: bus cycle)
  - Fast-Page mode -multiple reads without RAS if same row
- DRAM optimizations

6) row buffer

- 7) if read other row, write back
- data into (slow)
- 4) choose column
- 5) read data
- 3) CAS signal
- Read cycle
- 2) choose row
- 1) RAS signal
- read buffer (fast)
- into buffer (slow)
- row buffer
- DRAM row/page

DRAM (used in main memory)

- Read destructive
- upper half (RAS Row Access Strobe)
- ^/2 address line 地址线复用
- lower half (CAS Column Access Strobe)
- need refresh 刷新 * 异步刷新 -各行分布 refresh cycle
- characteristics
  - volatile
  - density first
  - 1 transistor + 1 capacitor per bit
  - 存储矩阵 memory array
    - 2^ units
    - ^ address
    - 2^⌈^/2⌉ address 双译码写
    - column ↑
    - bus ↓
    - row

SRAM used in cache

- characteristics
  - volatile
  - 6 transistors per bit
  - speed first density second

CPU-memory gap

- $S_{CPU} > S_{mem}$
- How to overcome
  - 顺序执行
    - 过程调用 Depth ≤ 5
    - 循环结构 -少数指令 -多次执行
  - operations on data structure 局限在小范围内
    - 局部性 Locality 原理
      - 时间 Temporal - recent access
      - 空间 Spatial - Access nearby address

Hierarchy

speed, unit cost ↑ / size ↓

- CPU registers
- Cache (SRAM) } Internal Memory 内存
- Main memory (DRAM) }
- Harddrive (SSD) } External memory 辅存
- Harddisk

CPU ↓↑ memory controller → DRAM Refresh
memory controller → Row Buffer management
memory controller → Address Mapping
memory controller ↓ request scheduling
memory controller → commands
memory channels ↑↓ data
DIMM ← memory channels

* one bank a time (independent)
* banks in a rank share address/control pins

xi = i banks + 1 ECC
↑ error-correcting code memory
2 ranks
↑
Dual in-line memory module
↑ DIMM

1 controller
1 channel
↑
1 controller
2 channels ← memory channels ← DRAM organization

2 controllers ↓ ↑ channels → ↑ bandwidth
2 channels

$t_a = H \times t_A + (1-H) \times t_M$; $H = \frac{\text{cache access no.}}{\text{cache access no.} + \text{memory access no.}}$

命中率 Hit rate
Block 大小 unit for internal memory ↕ cache

(what to do if full)
Placement Policy ↑
Write Policy ← (how to ensure cache and memory are the same)
Design → Address mapping (where)
Design → Data identification (how to find)
↑
average access time

Terms → Hit 命中 (Access data is in cache)
Terms → 未成功 Miss (Access data not in cache)

Cache (SRAM) - near CPU, hardware manage → 思想
思想 → we hope most data we access in cache
思想 → CPU search cache first
↓
store frequently access data in cache

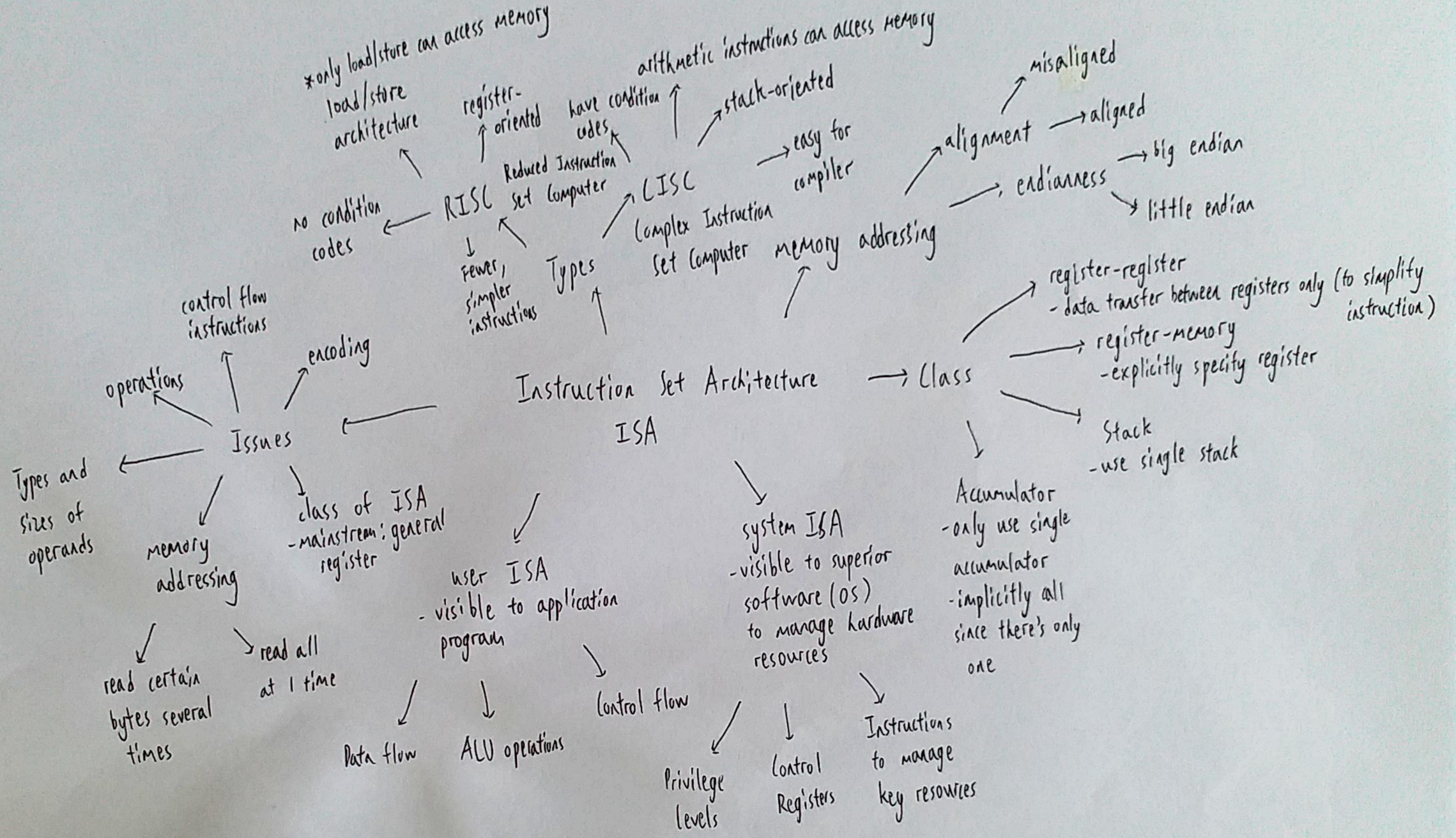
Memory 2 → Cache (SRAM)
Memory 2 → $N_{CPU} \times \Lambda_{CPU}$
Memory 2 → $N_{mem} \times \Lambda_{mem}$
Memory 2 → Memory Extension → word extension - extend address bus and capacity
Memory 2 → Performance Metrics
Memory Extension → Bit extension (extend data width) - parallel memory array

word extension
↓
address → total ⇈ $\log_2 N_{CPU}$
address → chips ⇊ $\log_2 N_{mem}$
address → decoder $k \to i$ → no. of chips
decoder → total - chips

no. of chips = $\frac{\Lambda_{CPU}}{\Lambda_{mem}}$
↓ data lines distributed to each chip
↓ address line directly connect to every chips
$= \log_2 N_{CPU}$

Performance Metrics
↓ Latency 延迟 - time per word
↓ Bandwidth 带宽 - data per unit time
↓ Aspects
Aspects → DRAM bank latency
Aspects → controller → DRAM
Aspects → DRAM → CPU
Aspects → Latency of memory controller
Aspects → (CPU → memory controller)
Latency of memory controller → request → instructions
Latency of memory controller → queue

Access time < Cycle Time
Access time: request → data
Cycle Time: interval between 2 requests


*only load/store can access memory
load/store architecture
register-oriented
have condition codes
arithmetic instructions can access memory
stack-oriented
RISC
Reduced Instruction Set Computer
CISC
Complex Instruction Set Computer
easy for compiler
no condition codes
fewer, simpler instructions
Types
memory addressing
alignment
misaligned
aligned
endianness
big endian
little endian
control flow instructions
operations
encoding
Issues
Instruction Set Architecture
ISA
Class
register-register
- data transfer between registers only (to simplify instruction)
register-memory
-explicitly specify register
Stack
-use single stack
Accumulator
-only use single accumulator
-implicitly all since there's only one
Types and sizes of operands
memory addressing
class of ISA
-mainstream: general register
user ISA
- visible to application program
system ISA
-visible to superior software (OS) to manage hardware resources
read certain bytes several times
read all at 1 time
Data flow
ALU operations
Control flow
Privilege levels
Control Registers
Instructions to manage key resources

stack

high address — fp

low address — sp

stack canary → avoid such codes

user level

Buffer overflow prevention

system level

non-executable code segments

randomized stack offset

- store address of next instruction in ra
- jal procedure address

jump and link (J-type)

2) jal caller → (control) callee

1) caller put argument in register a0~a3/stack

3) callee allocate resources

4) callee do work

5) callee store return value in v0/v1

6) callee → (control) caller jr ra

- a0~a3
- ra, sp, gp, tp
- s0~s7

store when call function

2 register types

don't store when call function

- return value register v0, v1
- temporary register t0~t9

function call steps

R-type

MIPS

Microprocessor without Interlocked Piped Stages

Properties

need compiler support
- rearrange sequence, reduce pipeline conflict

load/store architecture
- only load/store can access memory

more registers

little type of operand addressing

- register addressing
- immediate addressing
- base address + 16 bit offset

fixed length : 32bits

simple → fewer instructions, simpler function

simple format

conditional transfer instructions (I-type)

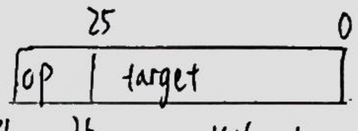
3) +PC(+4) ← next instruction address

2) sign extension → 32 bit

$-2^{15} \sim 2^{15}-1$ word range ← relative offset → 1) 16 bit offset << 2 .. 18 bit

J-Type (Jump)

absolute offset → 1) offset << 2, become 28 bits; 2) higher 4 bit of PC add to front of offset become 32 bits

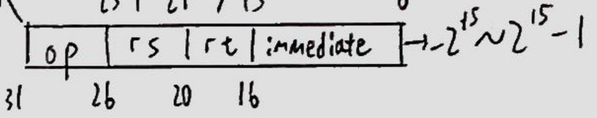


unconditional

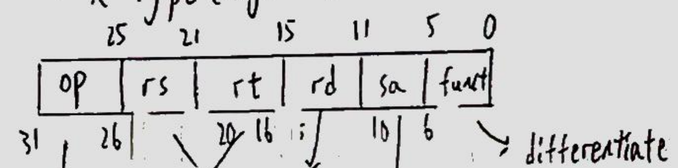
3 instruction format
* higher 6 bits to identify

I-type (immediate) → load/store target address = 32 bit address + 16 bit offset

differentiate function — operand register — target register



R-type (register)



Intel x86 ISA

- Instructions
  - Jump
    - unconditional JMP D
      - 3B offset NEAR
      - 2B offset SHORT
    - conditional jump: jcc D ← condition
  - arithmetic operation
    - 包括结果调整，符号扩展
    - single operand: neg / not; dec / incq
    - double operand
      - 不允许 2 operand from memory
      - add/sub: same for signed/unsigned
  - transfer type
    - 不允许 memory → memory
    - q=64 / l=32 / w=16 / b=8
    - Addressing general way
      - D(Rb, Ri, S)
      - offset (1/2/4B)
      - base register
      - index register (any register)
      - scale (1/2/4/8)
      - = Mem[Reg[Rb] + S×Reg[Ri] + D]
    - x86-64
      - use 页式存储
      - 多级页表映射
      - GDTR[CS] + RIP
      - = logical address
- Properties
  - use CISC
    - more, complex instruction
    - different length, format
  - backward compatible
    - directly expand register
    - x86-64 has 8 new 64 bit general register
- addressing mode
  - 8086 实模式
    - PA = CS<<4 + IP
    - can directly change segment register value
    - no operating system, no 权限控制
    - unsafe way to access memory
  - 80386 or above 保护模式
    - safe, efficient
      - dynamically grow/link
    - to access memory → segment address (starting address of PA)
      - authority to access segment (read/write/execute)
      - size of segment
      - store in 段表 (memory)
        - GDTR
        - GDTR[CS>>3] + EIP = PA
- registers
  - general registers
    - data registers
      - AX : for */
        - AH
        - AL
      - BX : for offset
      - CX : counter
      - DX : part of multiplication (dividend)
    - pointer/index
      - SI : source
      - DI : destination
      - SP
      - BP
        - stack pointer
  - segment register
    - CS
    - SS
    - ES
    - DS
  - flag register
    - condition flag
      - CF
      - PF
      - AF
      - ZF
      - SF
    - control flag
      - TF : trace
      - IF : interrupt
      - DF direction

路↑，组↓，关联度↑

*if 组=1 → fully-associated

↑

关联度

↓

路↓，组↑，关联度↓

*if 路=1 → direct map

main memory

分 n 区

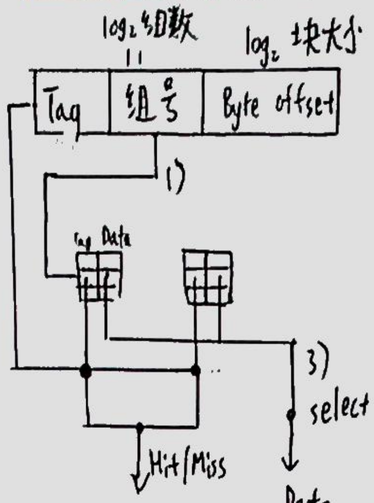
↓

区分 n 组 (direct map)

↓

组分 n 块

fully-associated

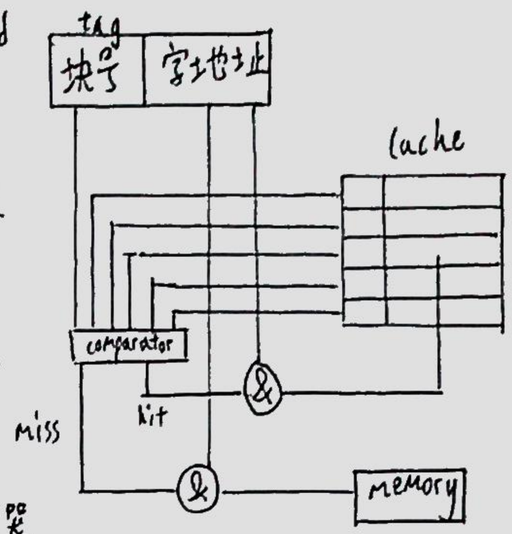


1) 组号 Direct mapped to cache

2) Tag → compare with rows in 组

3) tag same, hit

n-way → 1 data



行数 = Total capacity / Data bus 组相联 set-associated

组数 = Data bus / way

相联存储器
- use content as address to access

全相联 Fully-associated mapping

↑ cache use rate

conflict only if full

one-to-many

advantages → hit rate ↑

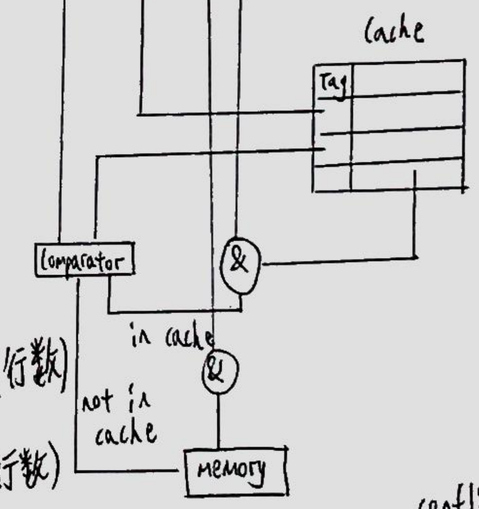
disadvantages → hardware cost ↑ (complex ↙)

Memory 3 → Address Mapping (到Cache)

Direct Mapped 直接 → Main memory divided to n 区 (same size with cache)

Tag blockid

区号 | 块号 | 字地址 | 1 valid bit



advantages
- one-to-one
- simple
- fast

disadvantages
- conflict ↑ → ↓ cache use rate
- low hit rate

storage

upper address into

highest hit rate

direct
- faster
hit rate ↓

fully/set associated
hit rate ↑

block id = $\log_2$(行数)

tag = B addr − ($\log_2$(行数) + ($\log_2$(word in block) + 2))

offset within B

1) use block id to find which block

2) use tag to find which line

3) use offset to find which word

Memory 3 → Address Mapping (到Cache)

Cache Write Policy

miss

-directly change memory

No-write-allocate *security

write-allocate *performance 按写分配法

Data → cache change in cache

good for 连续访问

hit

write-back
- write row in cache
*need dirty bit - check if changed
↓ if cache refresh
write memory

write-through 写直达
- directly write memory + cache

Cache 替换策略

O(n)
- uses binary tree 的LRU

LRU
Least recently used

FIFO

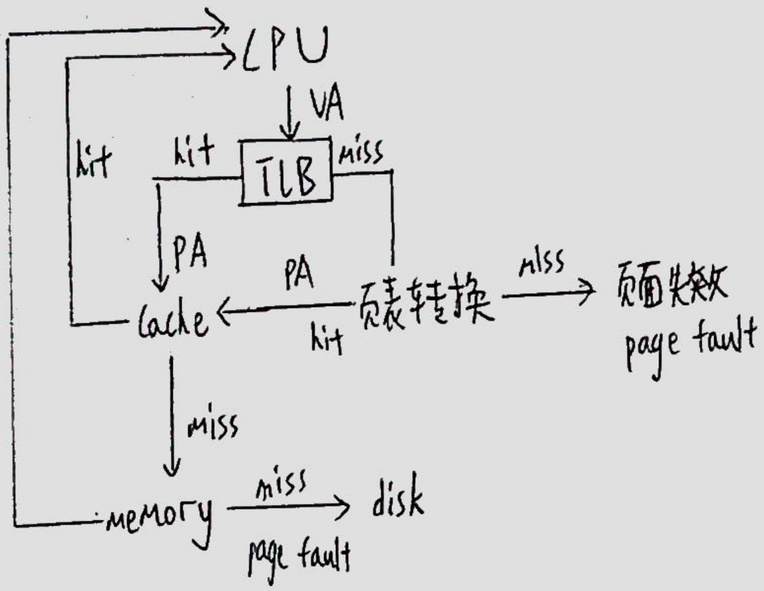
随机替换

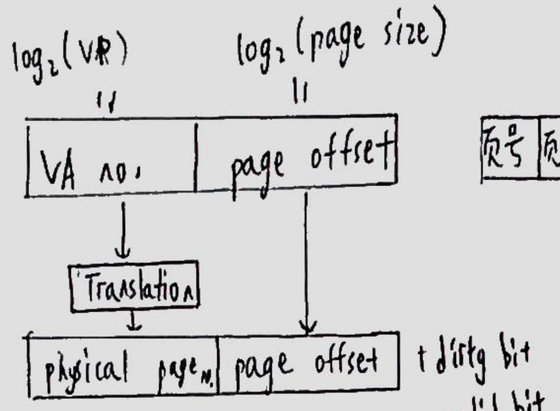
硬件实现

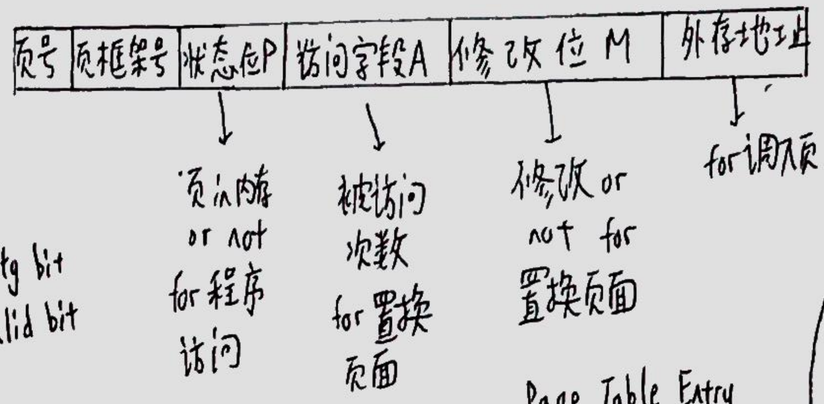
$\log n! \approx O(n \log n)$

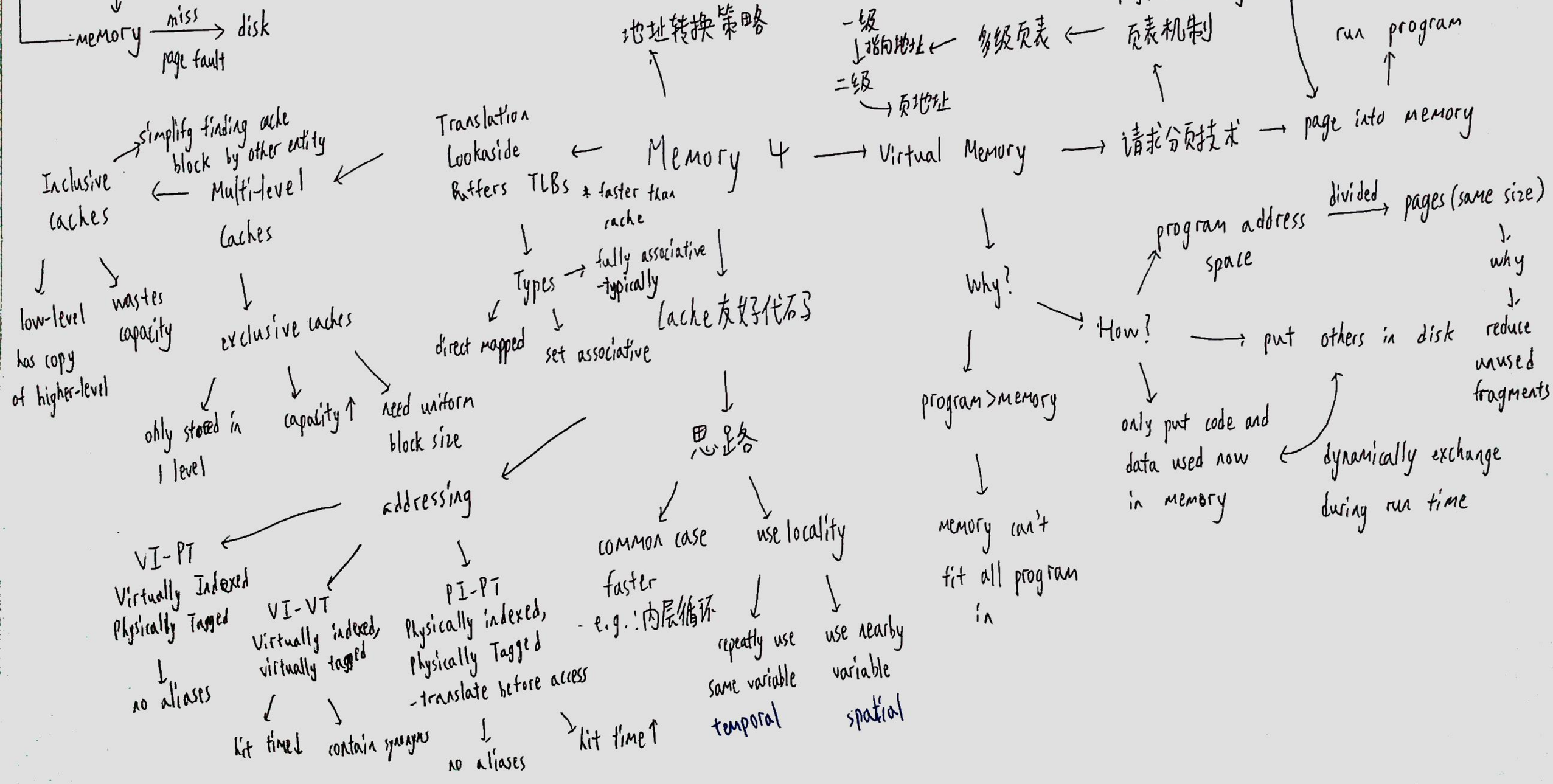
cache access time

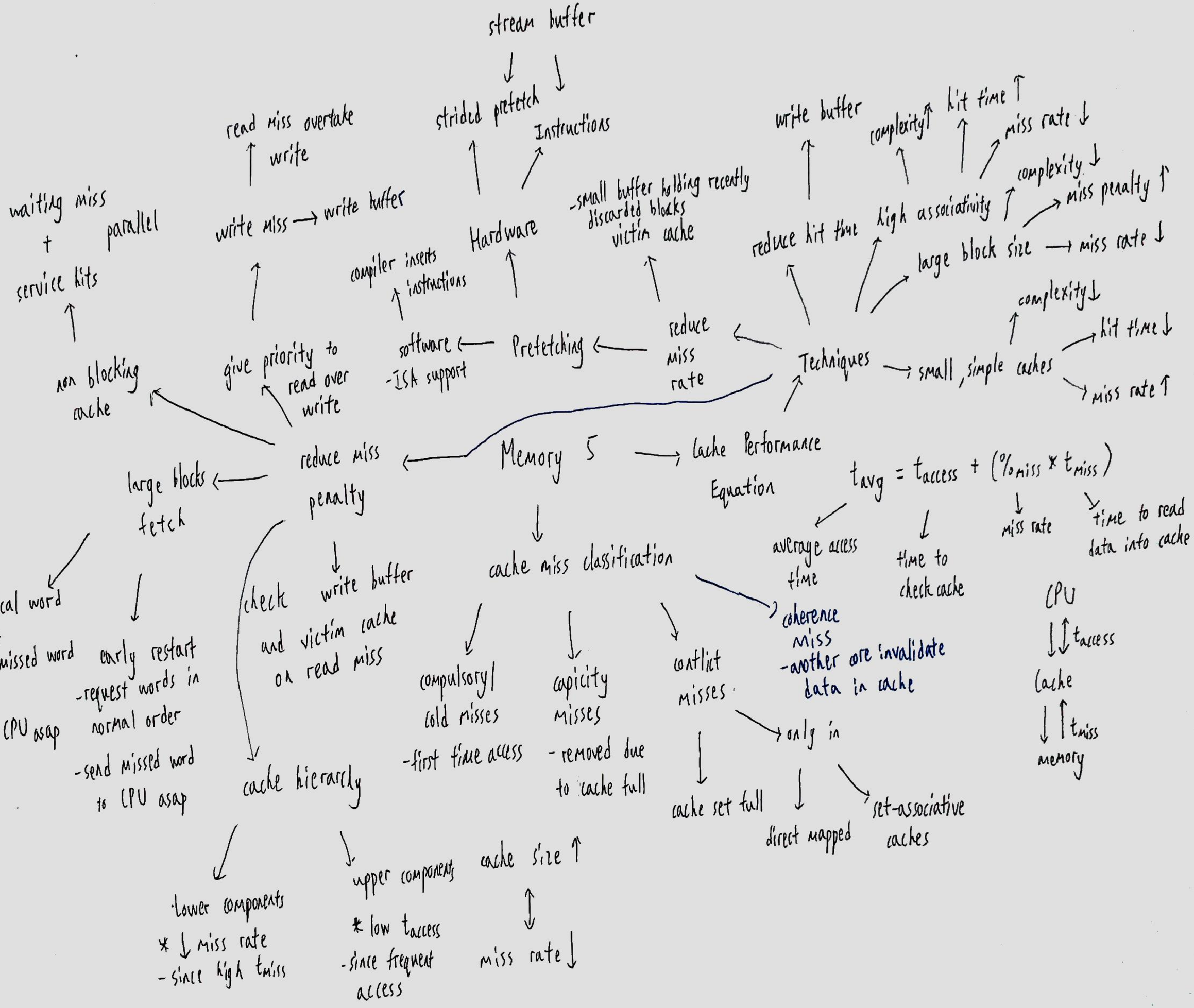
状态位
- 记n行访问顺序


CPU
VA
hit
hit
TLB
miss
PA
PA
Cache
页表转换
miss
hit
页面失效
page fault
miss
memory
miss
disk
page fault


log2(VR)
log2(page size)
VA no.
page offset
Translation
physical page no.
page offset
dirty bit
valid bit


页号
页框架号
状态位P
访问字段A
修改位 M
外存地址
页in内存 or not for程序访问
被访问次数 for置换页面
修改 or not for 置换页面
for调入页


Memory 4
地址转换策略
Translation Lookaside Buffers TLBs
* faster than cache
Types
fully associative -typically
direct mapped
set associative
Cache友好代码
思路
common case faster
- e.g.: 内层循环
use locality
repeatly use same variable
temporal
use nearby variable
spatial
addressing
VI-PT
Virtually Indexed Physically Tagged
no aliases
VI-VT
Virtually indexed, virtually tagged
hit time↓
contain synonyms
PI-PT
Physically indexed, Physically Tagged
-translate before access
no aliases
hit time↑
Inclusive caches
simplify finding cache block by other entity
Multi-level Caches
low-level has copy of higher-level
wastes capacity
exclusive caches
only stored in 1 level
capacity↑
need uniform block size
多级页表
一级
指向地址
二级
页地址
Page Table Entry
页表机制
Virtual Memory
Why?
program>memory
memory can't fit all program in
How?
program address space
divided
pages (same size)
why
reduce unused fragments
put others in disk
only put code and data used now in memory
dynamically exchange during run time
请求分页技术
page into memory
run program
used page not in memory
缺页中断请求

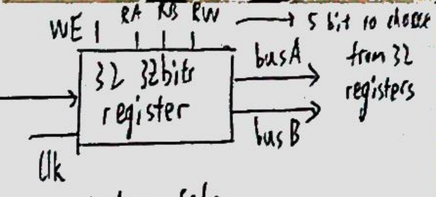

stream buffer
strided pretetch
Instructions
read miss overtake write
write miss → write buffer
waiting miss + parallel service hits
compiler inserts instructions
Hardware
-small buffer holding recently discarded blocks victim cache
write buffer
complexity↑ hit time↑
miss rate↓
complexity↓
miss penalty↑
reduce hit time
high associativity
large block size → miss rate↓
complexity↓
hit time↓
miss rate↑
non blocking cache
give priority to read over write
software -ISA support
Pretetching
reduce miss rate
Techniques → small, simple caches
reduce miss penalty
large blocks
fetch
Memory 5
Cache Performance Equation
$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$
miss rate
time to read data into cache
average access time
time to check cache
critical word first
- request missed word first
- send to CPU asap
early restart
- request words in normal order
- send missed word to CPU asap
check write buffer and victim cache on read miss
cache miss classification
coherence miss
- another core invalidate data in cache
compulsory/ cold misses
- first time access
capicity misses
- removed due to cache full
conflict misses
only in
cache set full
direct mapped
set-associative caches
CPU
t_{access}
Cache
t_{miss}
memory
cache hierarchy
lower components
* ↓ miss rate
- since high t_{miss}
upper components
* low t_{access}
- since frequent access
cache size ↑
miss rate ↓

Processor Design 处理器设计

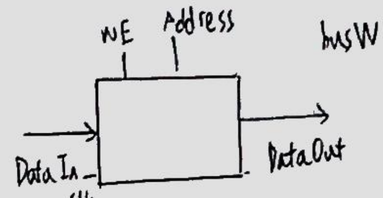
Cycles
- machine cycle / CPU cycle
  - formed by several clock cycles
  - shortest time to read 1 instruction from memory
- one basic instruction Clock cycle
- instruction cycle
  - time for getting instruction and run it
  - formed by several CPU cycles
- disadvantages
  - one components can only be used once per cycle
    - need more components
      - wasting chip area
  - bottleneck: cycle time = slowest instruction time
    - wasted time for faster instruction

single cycle processor
- one cycle per instruction
  - advantages
    - simple to implement
  - disadvantages
    - latency (register delay)
- multiple cycles for 1 instruction
  multiple cycle processor
  - advantages
    - can repeatly use same components for 1 instruction

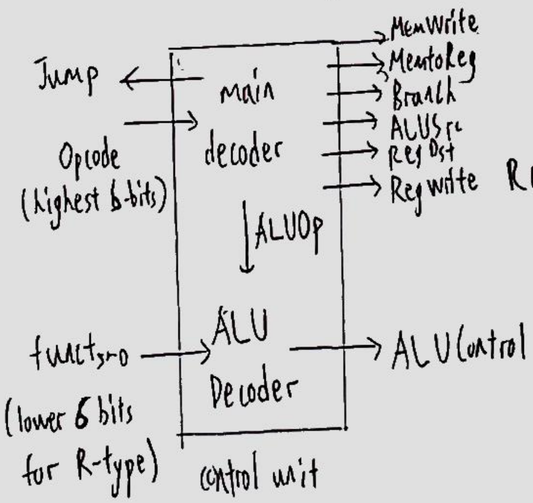
Steps
1) analyze ISA, get bus requirement
   - MIPS
     - load/store
     - ori
     - add/sub
     - branch
     - jump
2) choose suitable components for bus
   - register file 寄存器文件



   - 状态单元 synchronous sequential logic
     - edge-triggered (data sampled at clock edge)
     - memory
   - combinational logic
     - output only determined by input
     - MUX
     - ALU
     - Decoder



3) connect components by bus
4) analyze implementation of every instructions to determine control signals
   - MemtoReg
   - MemWrite
   - Branch
   - RegWrite
   - RegDst
   - ALUSrc
5) integrate control signal in one controller



hard-wired controller vs microprogramming control
- hard-wired controller
  - properties
    - fast
    - complex network
    - no rule
    - can't modify ISA/instructions
    - difficult to design/debug
- microprogramming control
  - 1 instructions = 1 microprogram
  - properties
    - easier to fix bugs/updates
    - create complex instruction without changing bus
  - many microinstructions
    - 1 bit ↔ 1 control signal

put the "=0?" test into ID phase

2 cycles faster

put an adder in ID phase
-calculate target address

-put several must executed instructions after branch

branch slot (compiler)

flush (delete readed instruction)

solution

分支预测
减少

2bit control signals + 32bit x2 + 5 RT/RD selection code
=
71 bits

ALUout

MEMout

MEM/WB segment register

4-bit control signal + 1 bit zero flag + 32bit x3 + 5-bit RT/RD selection code
=
106 bits

ALUout

branch address

RT value

EX/MEM segment register

memory operation

4) MEM

WR

5) write

3) Execution EXE

ID/EX segment register

main controller generates → 4-bit control signal

+

32 bit × 4 PC+4

immediate

RS

RT

+

5 bit × 2 RT, RD selection

=

147 bits

-need specific stall controller to check stall (hardware)

前向传递
hardware forwarding
- pass result asap
*need special circuit

-need results of other instructions to determine branch destination (control hazards)

solution

compiler (software)
-insert NOP

Data hazards
- use data which currently processed by other instructions

Hazards
冒险

流水线处理器
pipeline processor

Implementation
-use segment register
*they pass unused CS to next SR

2) Instruction Decode
ID

1) Instruction Fetch
IF

IF/ID segment register

32-bit instruction
+
32-bit next instruction address
=
64-bit

load-use data hazard
-read data when loading from memory hasn't finish

read after write
-read when writing hasn't finish

Structural Hazards
-needed hardware is used by other instructions

MIPS instructions are easy to be pipelined

reasons

same length: 32 bit

less format: 3

at most write 1 result

only load/store can access memory

insert nop, delay conflicts

solution

increase no. of hardware

use different memory for data and memory

pipeline certain function unit

* for register file
- 1 cycle for read and write

reset SR(EXL)
-enable interrupt
EPL → PC

CPU → kernel state

2) execute instruction from PC

1) execute eret

2) set STATUS SR(EXL) → forbid interrupt

3) set cause

5) return

1) set PC → EPC

4) get handle code from a port

handle steps

- 8-bits IM (Interrupt Mask)

STATUS

MIPS → registers → cause
-store cause of exception

EPC (exception program counter)
-store address of instruction which triggered exception

- interrupt vector

device → CPU x86

Exceptions / Interrupts

Pipeline Processor

external interrupts override others

only handle at commit point

flush all previous instructions

- set a commit point (WB)

record exceptions flags in pipeline (earlier override later)

Precise

Interrupt / Exception in pipeline

Imprecise → can't pinpoint which instruction triggered exception

handle asap

when handling, flush other possibly simultaneously occured exception instruction

arithmetic overflow → EX

undefined instructions → ID

Types

TLB miss / page fault

MEM IF

I/O service request → any

any ← hardware malfunction

4) restore state

3) handle interrupt

2) save state

general handle interrupt steps

1) switch control

Types

internal

external → clock interrupt

keyboard / mouse

shut down / restart

self

forced

divide by 0

software exception

overflow

out of bound

instruction
-int type

program

request

OS

hardware malfunction

optimization

execution time = no. of instructions x CPI x clock cycle

↓ CPI (multiple issue)
- multiple instruction parallelly executed

dynamic
-at run time

out of order

in order

static
- compiler decide

↓ clock cycle (superpipelining)
- increase no. of pipelines

disadvantages

↑ complexity,
↑ costs

↑ correlation,
↑ delay by stall

↑ power

delay of segment register

Fetch → Decode → Rename → Dispatch

In-order front end

↓

Window of instructions

Issue → Reg-Read → Execute → Writeback

Out-of-order execution

↓

In-order commit

handle at commit point

precise interrupt

nop

true

false

register renaming

RS's registers
-better than compiler renaming

WAW 输出相关
write after write

rename registers as value/pointer to RS

↓

avoids WAR/WAW

every Function Units have buffer

↓ reservation station RS

Tomasulo Algorithm
-high performance without special compilers

↓ use common Data Bus CDB to broadcast results

Stages

1) Issue: get instruction from FP Op Queue
   -if RS free, rename registers
2) Execution: operate on operands
   -if not ready, wait CDB
3) Write result: write to CDB, free RS

↓

Precise Exceptions
-Reorder Buffer ROB

Speculative Tomasulo

1) Issue: send instruction and reorder buffer no.
2) execution: same
3) write result: write to CDB and ROB
4) commit: update register with reorder result according to ROB

Control Hazards ← problems → data corellation

BTB+BHT

↑ ↙

Use branch prediction

↓

Branch Target Prediction

↙ ↓ (decode)

Return Address Stack RAS

return

PC = RAS[-Top]

call

RAS[Top++] = P(t+4

Branch Target Buffer BTB
-store history as a table

* works for direct transfer only

↓

if an instruction = branch
1) renew BTB
2) predict using stored history

↓

Branch History Table BHT
↑ (calculate branch ~~target~~ outcome)

Branch Outcome Prediction

↓

Branch History Register BHR
-record recent branch outcome pattern

not jump -1 ← 2 bit → jump +1

T/NT
1-bit

initially choose BHT
↑
Chooser
↓ after threshold, choose Gshare

Gshare
PC xor BHR
↓
BHT

dynamic
-out-of-order

WAR 反相关
write after read

超标量 superscalar

static
-in-order

hardware
-dynamically select no. of instructions for issue

software
-compiler do its best to rearrange instructions

Multiple Issue Processor 多发射处理器

Very Long Instruction Word
VLIW
超长指令字
-pack 2 or more instructions into 1 VLIW

↓

use compiler

↑ expand loop
↑ Instructions
↑ ILP

↓ branch → cost ↓

advantages

↓ simple hardware

↓ cost

change instructions order

↓ correlation

register renaming

↓ correlation

limitations

↓ complex compiler

target code not compatible

expand code

↓

↑ storage

one instruction stuck, whole process stuck

↑

lock step

↑

Explicit Register Renaming
-use a physical register file > no. of registers
- map ISA register → physical register

↓ allow data fetch from single register file

↓ decouple renaming from scheduling

↓

precise interrupt

only reduce vertical waste (still exists)

when a thread stalls for a long time ↓ switch

switch per instruction

Coarse-Grain

Fine-Grain ← Vertical Multithreading

- multiprocessor Chip Multiprocessing CMP

need to ↓ context switch time

Multithreading → Costs → Cache / TLB miss caused by competition

Costs → OS

Purpose: limitation of Instruction Level Parallelism

each thread needs to store → system state → virtual memory page table base register

PC

↓ user state ↓ GPR

↓ exception handling registers

Out-of-Order superscalar

Simultaneous Multithreading SMT
- multithreading at same cycle

can simultaneously issue

↑ memory bandwidth and port ← hardware → need ↑ function unit

correlation → limited multiple issue

↓ delay caused by rearranging instructions

↓ ↑ register file bandwidth and port

↓ need ↑ instruction bandwidth

High thread level parallelism

HTLP

- one cycle many threads

Low thread level parallelism

LTLP

- one thread can use up whole width

vertical waste: idle cycle

↑

waste

↓

horizontal waste: partially filled cycle

Consistency

Memory consistency

- unordered load/store
Release consistency
↑ allow compiler optimizations

Total Store Order TSO
- store in order
↑ allow reading store buffer to bypass value

store buffer
- put all store into buffer to hide write latency

Sequential consistency SC
- load/store order same as ode for every CPU

Cache Coherency
- shared cache + private cache

- one or more CPU share same block,
- writing different part of it
False Sharing

consistency protocol
一致性协议

Modified/Exclusive/Shared/Invalid
- 1 copy
- up to date data

MSI
Modified/Shared/Invalid
- can read/write
- 1 dirtied copy

- can read
- multiple clean copy

Valid/Invalid VI
can't read/write

only allows one copy exists

Explicit parallelism
显式并行性
multi-threading

shared memory implementation

硬件多线程 hardware multithread
- 1 pipeline
- multiple PC and register files to save thread status
- switch thread

advantages
↑ throughput
↑ pipeline use rate

multi-core → multi pipeline
- 1 memory

物理复用同1处理器
multiplexed-pipeline
- 1 processor
- 1 pipeline
- context-switching