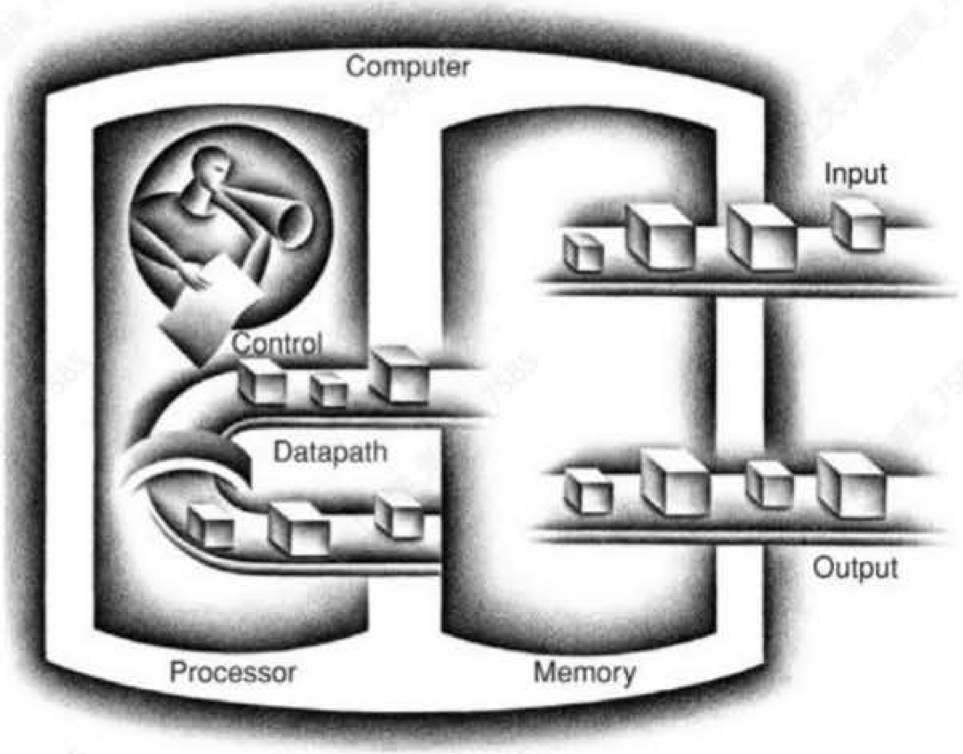
计算机体系结构期末课程梳理

潘哲

2021年12月24日

什么是 Computer Architecture?

- *Computer Architecture* is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance, cost and power goals.



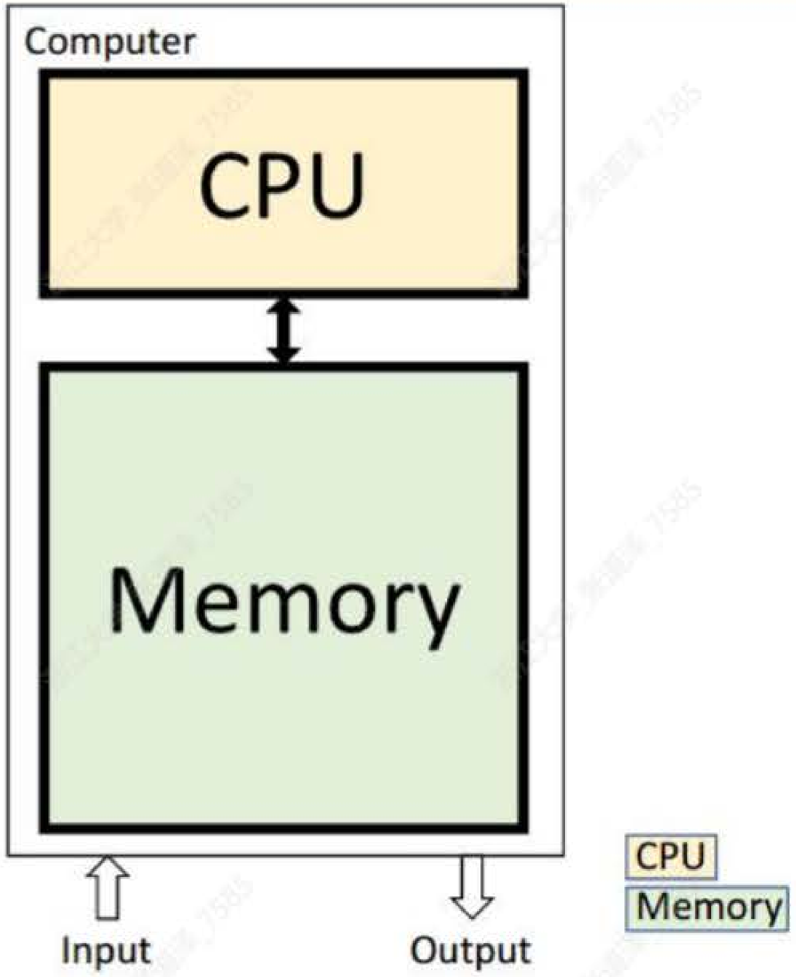
章节回顾

	Topic	Reference
Ch1.	Fundamentals of Quantitative Design and Analysis (Quantitative principles)	Chapter 1
Review	Pipelining: Basic and Intermediate Concepts (How to design a pipelined CPU)	Appendix C
Ch2.	Memory Hierarchy Design (How to improve cache and memory)	Chapter 2 Appendix B
Ch3.	Instruction-Level Parallelism and Its Exploitation (How to implement dynamic scheduling)	Chapter 3 Appendix A
Ch4.	Data-Level Parallelism in Vector, SIMD, and GPU Architectures	Chapter 4
Ch5.	Thread-Level Parallelism	Chapter 5

梳理

- 如何去公平的衡量一个体系结构设计？(Ch1)
- 如何去优化memory部分？(Ch2 Appx B)
- 如何去优化CPU部分? (Ch3, 4, 5 Appx C)

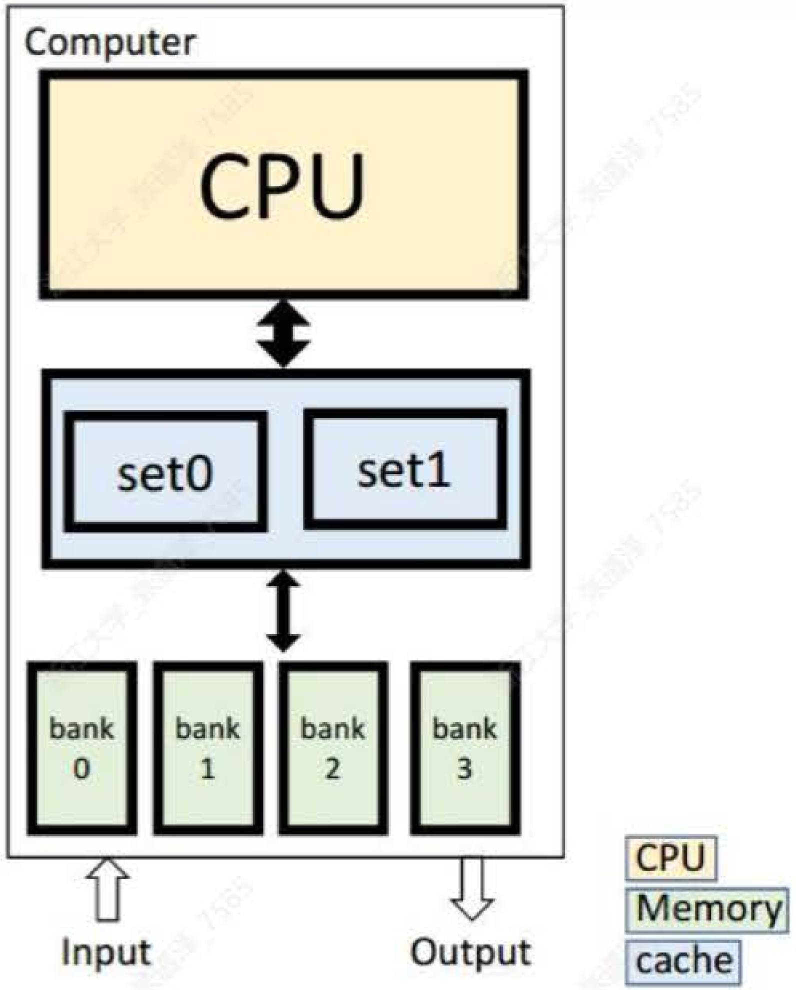
Answer：量化分析，对系统建模，抽象成小学应用题



梳理

- 如何去公平的衡量一个体系结构设计？ (Ch1)
- 如何去优化memory部分？ (Ch2 Appx B)
- 如何去优化CPU部分? (Ch3, 4, 5 Appx C)

Answer: 在CPU和memory之间增加cache， cache优化， memory分bank交叠

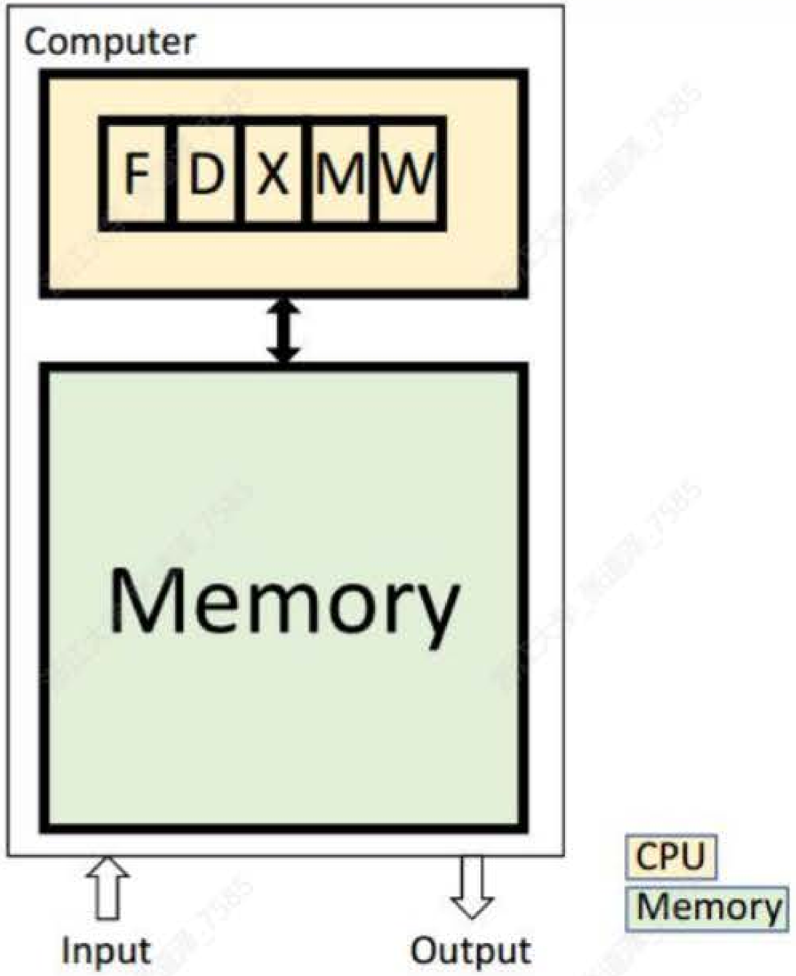


梳理

- 如何去公平的衡量一个体系结构设计？ (Ch1)
- 如何去优化memory部分？ (Ch2 Appx B)
- 如何去优化CPU部分? (Ch3, 4, 5 Appx C)

Answer: 流水线

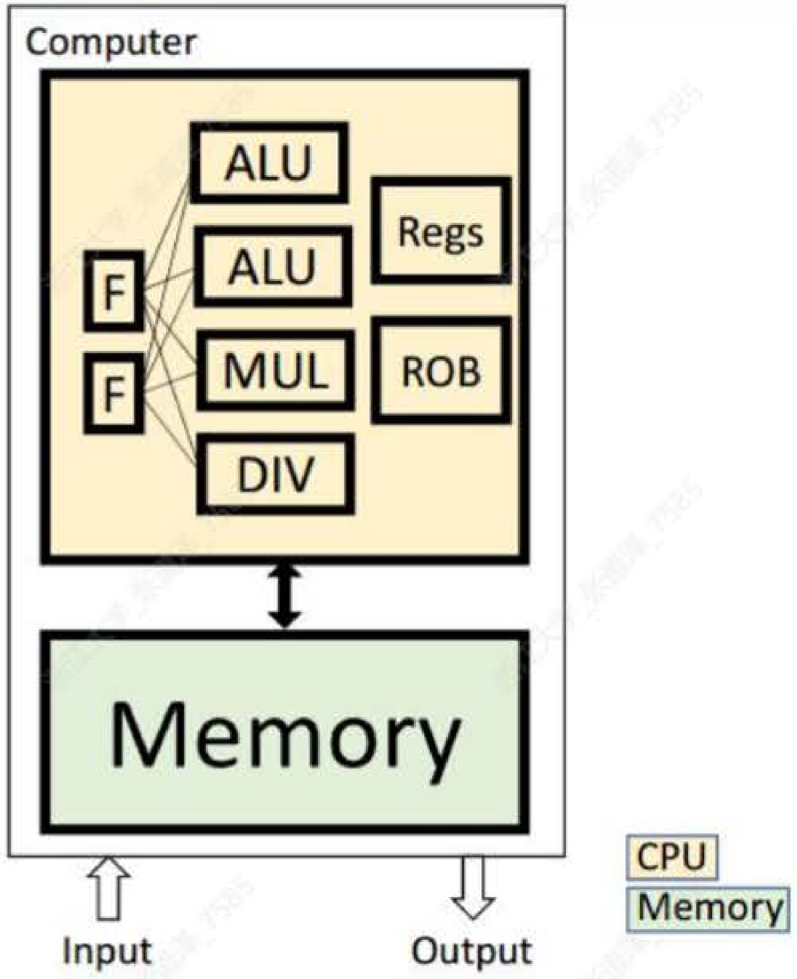
引入的问题：控制冲突，结构冲突，数据冲突



梳理

- 如何去公平的衡量一个体系结构设计？ (Ch1)
- 如何去优化memory部分？(Ch2 Appx B)
- 如何去优化CPU部分? (Ch3, 4, 5 Appx C)

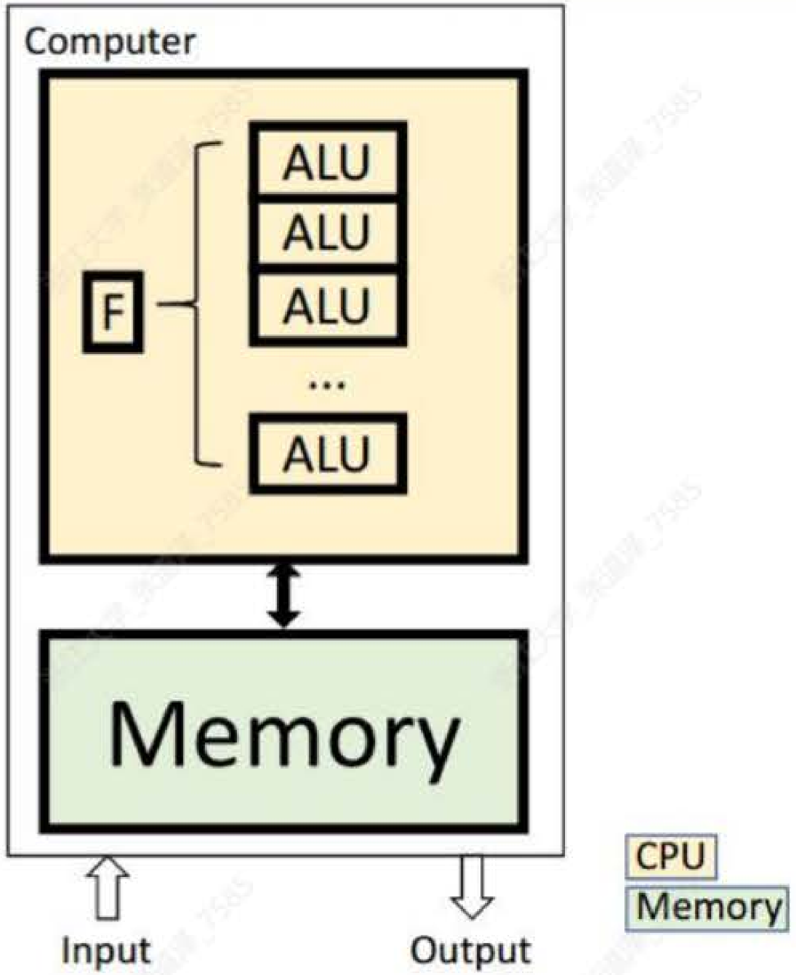
Answer: 流水线，动态调度，分支预测，投机，多发射，多线程，提高执行效率



梳理

- 如何去公平的衡量一个体系结构设计？(Ch1)
- 如何去优化memory部分？(Ch2 Appx B)
- 如何去优化CPU部分？(Ch3, 4, 5 Appx C)

Answer: 将运算单元复制多份，执行类似的任务，提高运算能力 SIMD,GPU

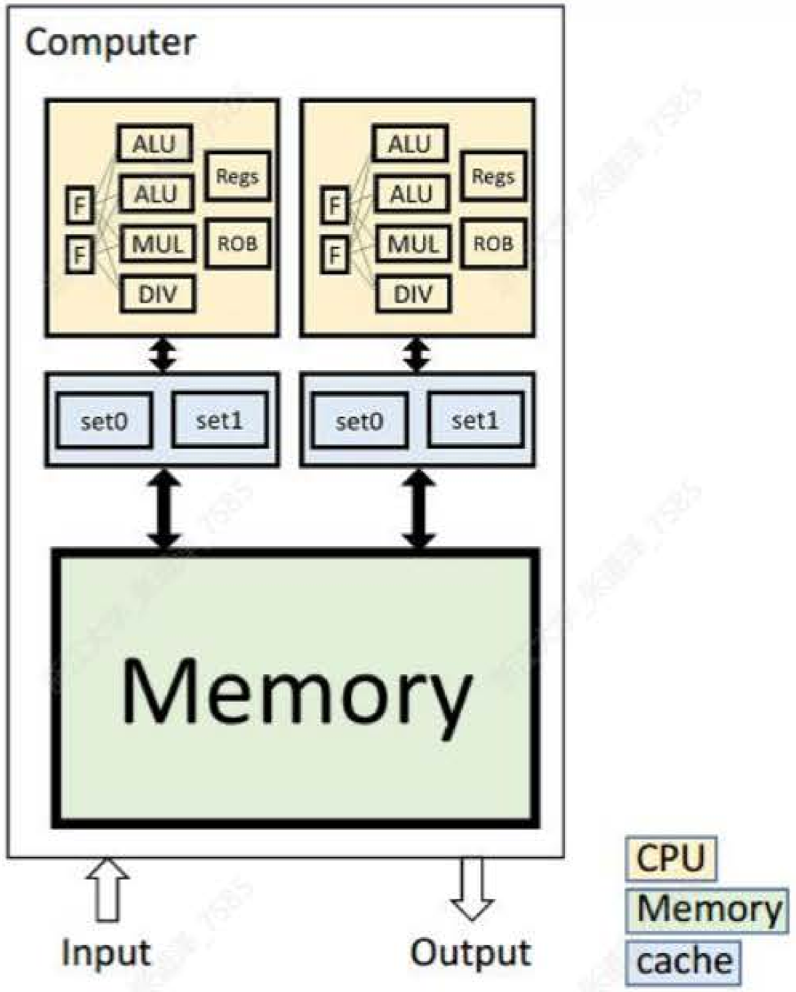


梳理

- 如何去公平的衡量一个体系结构设计？ (Ch1)
- 如何去优化memory部分？ (Ch2 Appx B)
- 如何去优化CPU部分? (Ch3, 4, 5 Appx C)

Answer: 多处理器， 多线程

引入的问题： cache一致性，memory连续性



第1章：量化设计与分析

Challenges of “three walls”

- ILP Wall
  - diminishing returns on finding more ILP HW (Explicit thread and data parallelism must be exploited)
- Memory Wall
  - growing disparity of speed between CPU and memory outside the CPU chip. Memory latency would become an overwhelming bottleneck in computer performance.
- Power Wall
  - the trend of consuming double the power with each doubling of operating frequency

第1章：量化设计与分析

- Dynamic power: power consumption in switching transistors.
  - $\text{Power}_{\text{dynamic}}$ = ½ *Capacitive load * Voltage^2 * Frequency switched
  - $\text{Energy}_{\text{dynamic}}$ = Capacitive load * Voltage^2

- Static power: power consumption when a transistor is off due to power leakage
  - $\text{Power}_{\text{static}}$ = current static * Voltage

第1章：量化设计与分析

Cost of an **Integrated Circuit**

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^N$$

第1章：量化设计与分析

Measurements of Dependability

- Module ***reliability***: continuous service accomplishment (of the time to failure)
  - MTTF: Mean Time To Failure
  - MTTR: Mean Time To Repair (service interruption)
  - FIT: Failure In Time = 1/MTTF
  - MTBF: Mean Time Between Failure = MTTF+MTTR
- Module ***availability***
  - $\frac{\text{MTTF}}{\text{MTTF + MTTR}} = \frac{\text{MTTF}}{\text{MTBF}}$

跟MTTF相关的不要想多了，直接求倒数就是损坏频率

第1章：量化设计与分析

Amdahl's Law

The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

第1章：量化设计与分析

Example Suppose that we want to enhance the processor used for web serving. The new processor is 10 times faster on computation in the web serving application than the old processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{\text{enhanced}} = 0.4$; $\text{Speedup}_{\text{enhanced}} = 10$; $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \dfrac{0.4}{10}} = \dfrac{1}{0.64} \approx 1.56$

- 建议
  - 记不住的性能相关公式抄到A4纸上
  - 审清题意，将问题退化成小学应用题，实在不会做假象一个值去推，如上题可假想原执行时间为1000s去推现执行时间

第2章：存储器层次结构设计

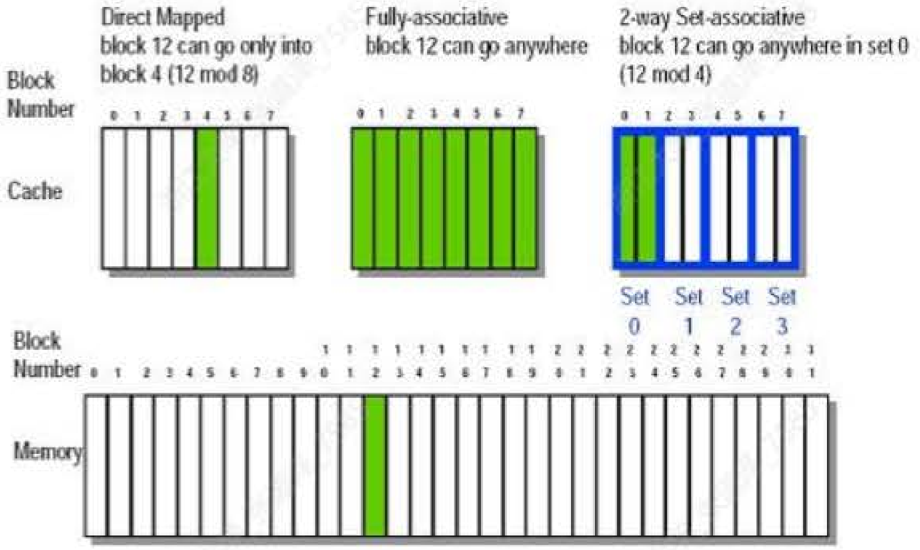
36 terms of Cache

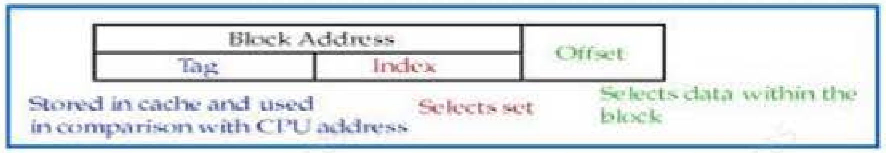
Cache	Virtual memory	
data cache	Instruction cache	unified cache
block	page	tag field
Block address	index field	block offset
full associative	set associative	**direct mapped**
n-way set associative	set	address trace
misses per instruction	Memory stall cycles	**miss penalty**
Valid bit	**dirty bit**	**locality**
cache hit	hit time	
cache miss	miss rate	page fault
Write through	write back	**write allocate**
random replacememt	least-recently used	**no-write allocate**
Average memory access time	write buffer	write stall

第2章：存储器层次结构设计

Four Questions for Memory Hierarchy Designers

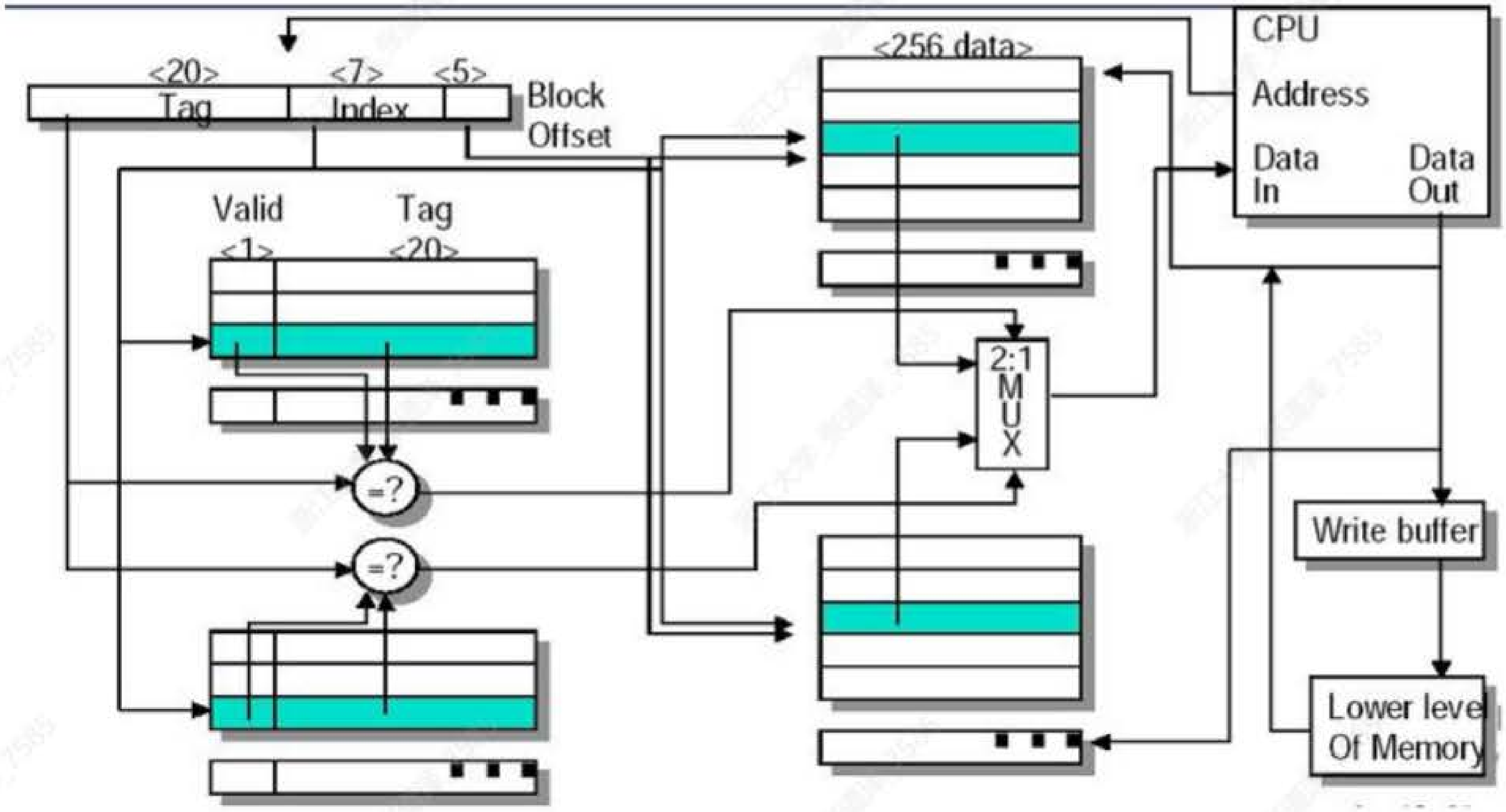
- Q1: Where can a block be placed in the upper level?
  (Block placement)
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the upper level?
  (Block identification)
  - Tag/Block
- Q3: Which block should be replaced on a miss?
  (Block replacement)
  - Random, LRU,FIFO
- Q4: What happens on a write?
  (Write strategy)
  - Write Back or Write Through (with Write Buffer)



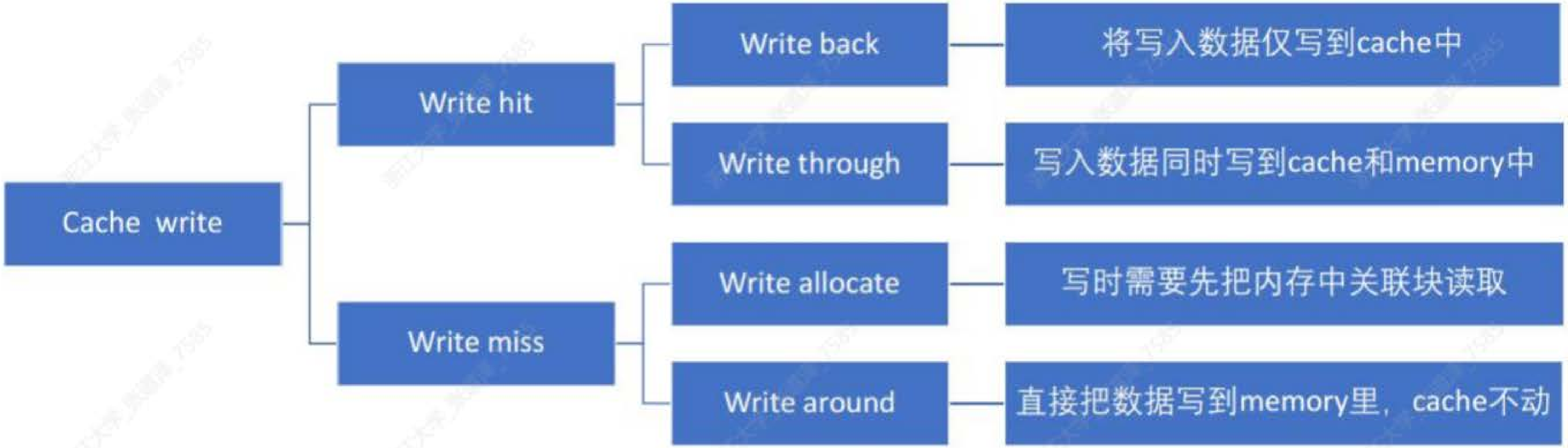

Block Address
Tag
Index
Offset
Stored in cache and used in comparison with CPU address
Selects set
Selects data within the block

第2章：存储器层次结构设计

- Memory size: 4G, Cache 8K, 2-way set associate



第2章：存储器层次结构设计



read/write　　miss/hit　　　dirty/clean　　自由组合生成8个case
自己推一下各自所需始终周期数。例如write back+write around策略下write miss dirty需要写回一个块到memory，从memory读一个块，最后再写到cache上

第2章：存储器层次结构设计

- Assume a fully associative wtrie-back cache with many cache entries that starts empty.below is a sequence of five memory operations(the address is in square brackets):

  1 write Mem[100];
  2 write Mem[100];
  3 Read Mem[200];
  4 write Mem[200];
  5 write Mem[100];

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer ：

for no-write allocate	misses:	1,2,3,5
	hit :	4
for write allocate	misses:	1,3
	hit :	2,4,5

第2章：存储器层次结构设计

How to Improve Cache Performance?

$$AMAT = HitTime + MissRate \times MissPenalty$$

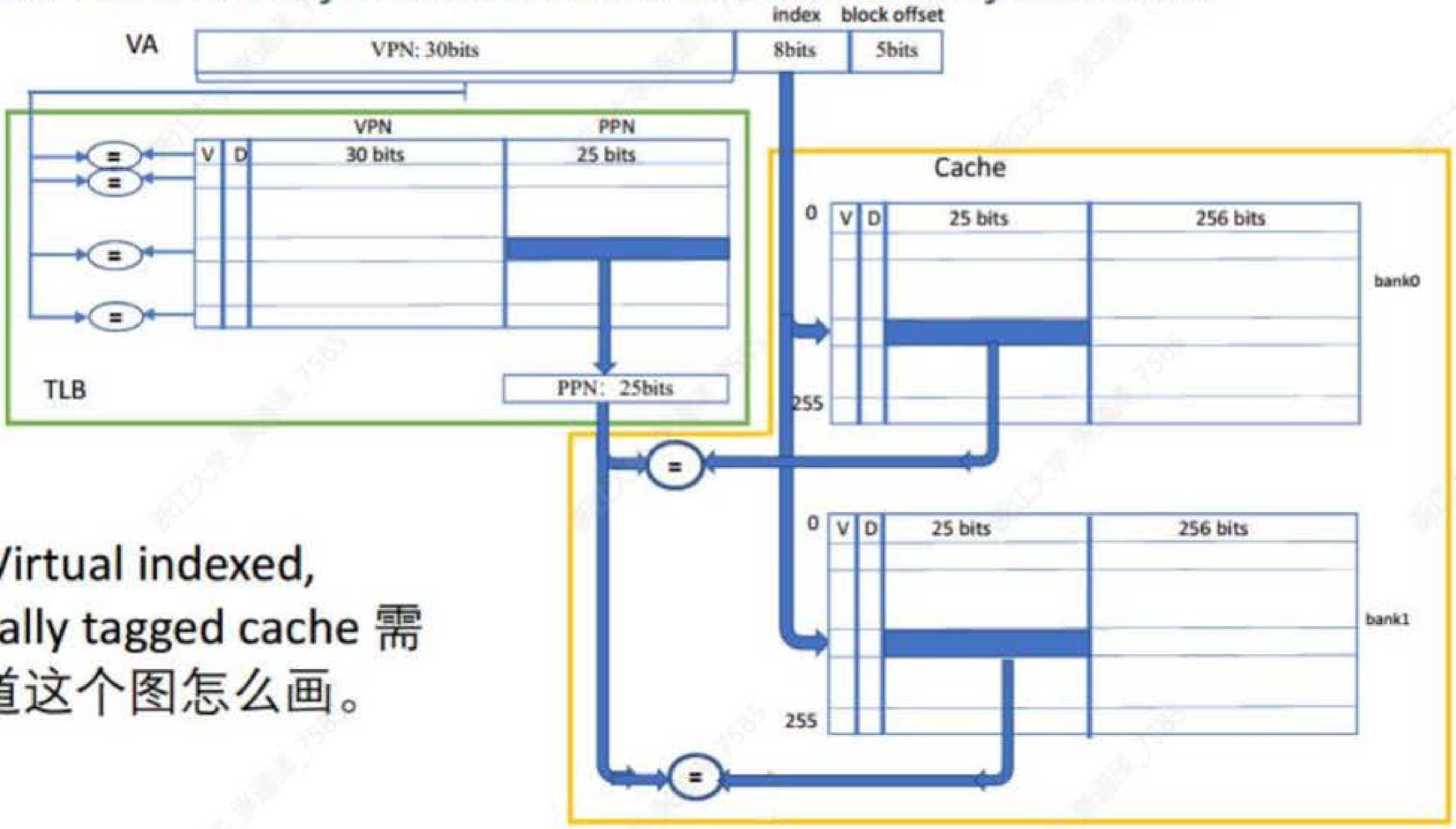
1. Reduce the time to hit in the cache.--4
   ——small and simple caches, avoiding address translation, way prediction ，and trace caches
2. Increase cache bandwidth .--3
   —— pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
   ——multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
   ——larger block size, large cache size, higher associativity,and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
   ——hardware prefetching,and compiler prefetching

Tip：

- 整页抄在A4纸上
- 计算题有空自己推一下
- 过程一定要写，至少有过程分。如果直接写答案错了没分
- 中间结果不要提前近似，一定要算到最后一步再近似

第2章：存储器层次结构设计

Virtual address wide = 43 bits, Memory physical address wide = 38 bits, Page size = 8KB.
Cache capacity =16KB. If a virtually indexed and physically tagged cache is used.
And the cache is 2-way associative write back cache with 32 byte block size.

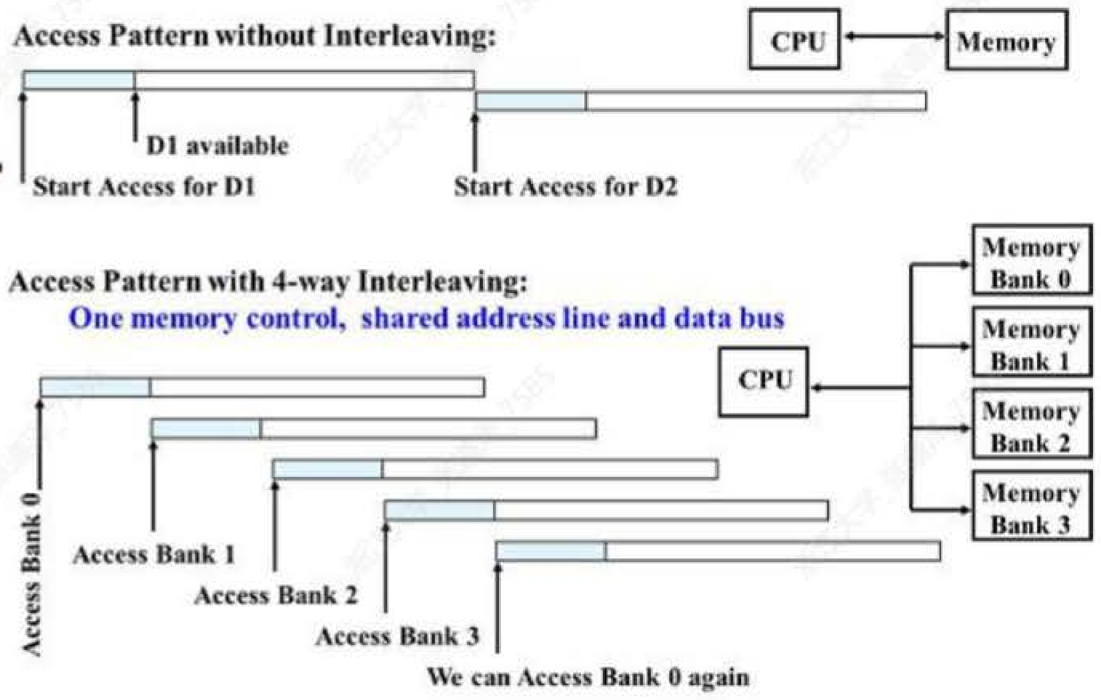


Tip： Virtual indexed, physically tagged cache 需要知道这个图怎么画。

第2章：存储器层次结构设计

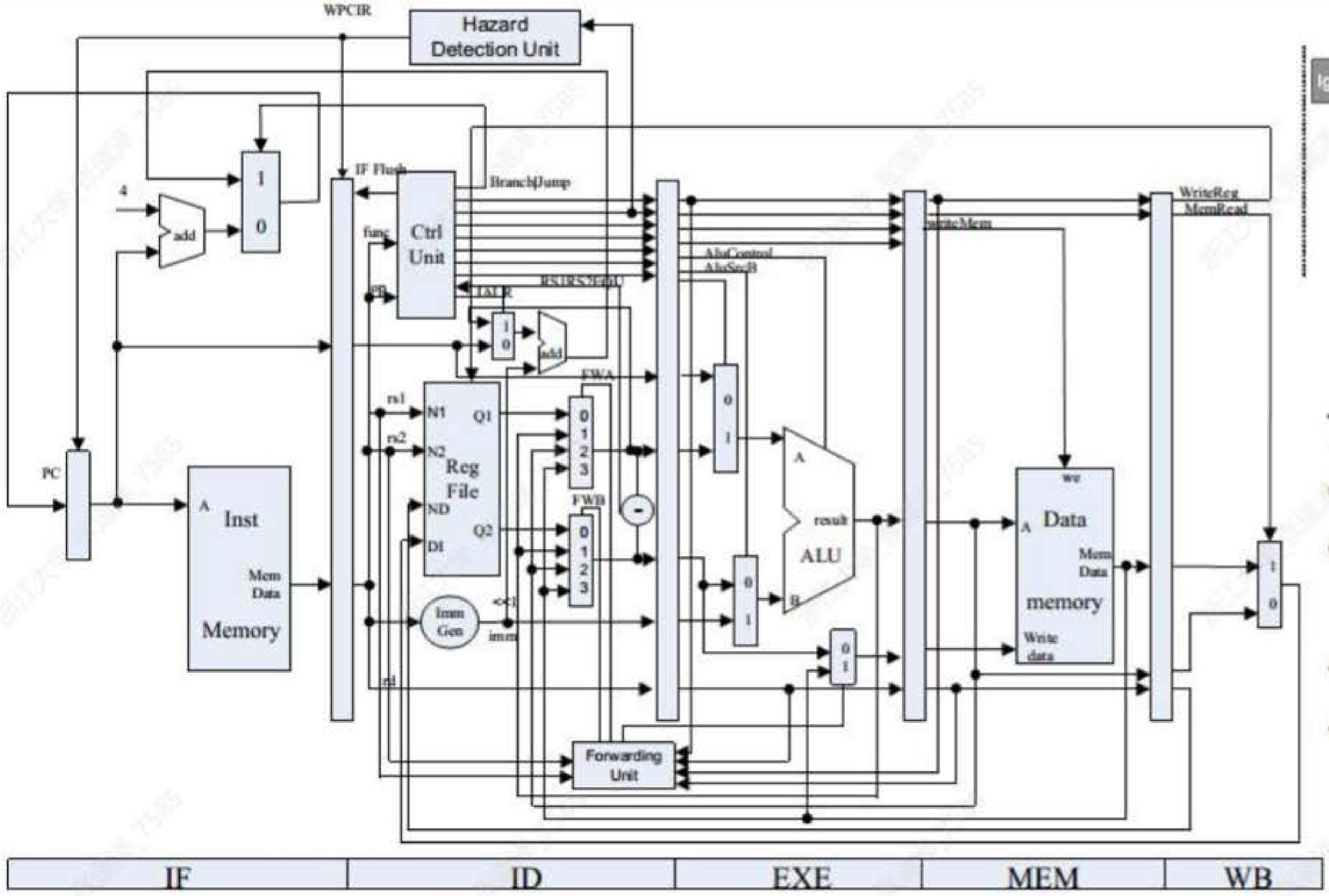
Performance of main memory

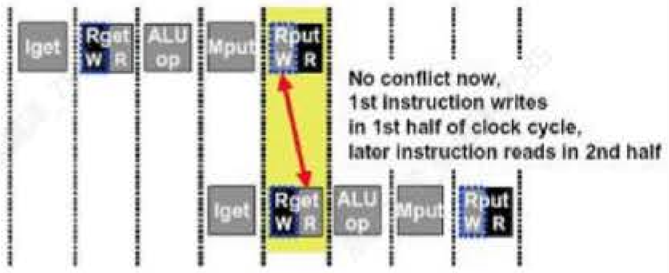
- *Latency*
  - harder to reduce latency ; *Important for caches.*
- *Bandwidth*
  - easier to improve bandwidth with new organizations
  - *Important for I/O.*
  - *Also for cache with second-level and larger block sizes.*



Tip：带宽，延迟和interleave的思想要懂，其他自己看

第3章：指令级并行性及其开发方法（pipeline）





Iget
Rget W R
ALU op
Mput
Rput W R
No conflict now, 1st instruction writes in 1st half of clock cycle, later instruction reads in 2nd half

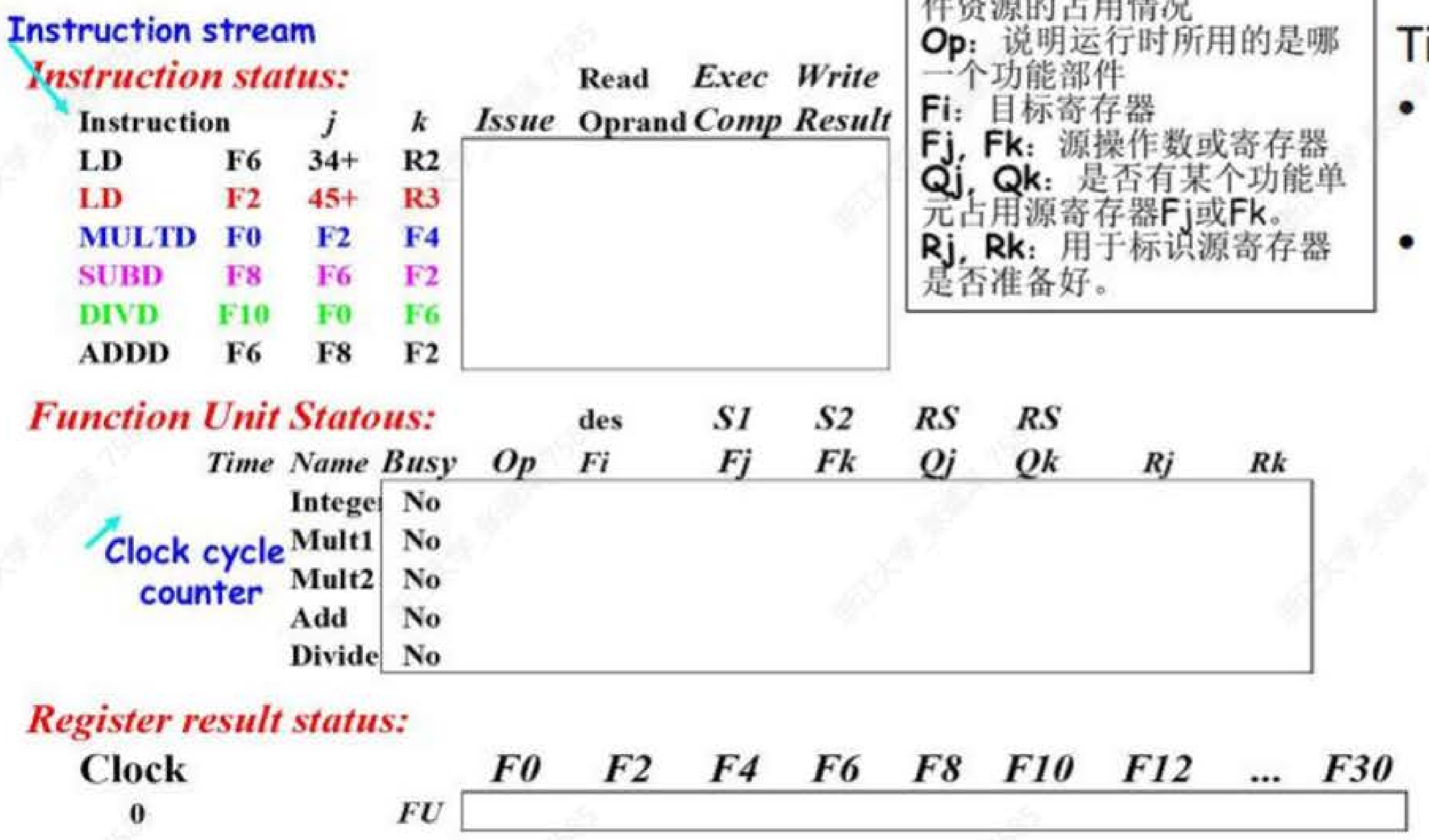
Tip：

- 回忆一下3个hazard
- 相邻两条指令冲突，没有forwarding停3个周期
- Forwarding总共有4条
- Lw的后有些情况forwarding不能解决

第3章：指令级并行性及其开发方法（Scoreboard）

- **Issue:** a instruction is issued when
  - *The functional unit is available and*
  - *No other active instruction has the* ***same*** *destination register.*
  - Avoid **strutural** hazard and **WAW** hazard
- **Read Operands** (RO)
  - *The read operation is delayed until both the operands are available.*
  - *This means that no previously issued but ncompleted instruction has the operand as its destination.*
  - This resolves **RAW** hazards **dynamically**
- **Execution (EX)**
  - *Notify the scoreboard when completed so the functional unit can be reused.*
- **Write result (WB)**
  - *The scoreboard checks for* ***WAR*** *hazards and stalls the completing instruction if necessary.*

第3章：指令级并行性及其开发方法（Scoreboard）



Instruction stream

Instruction status:

Instruction		*j*	*k*	*Issue*	Read Oprand	*Exec Comp*	*Write Result*
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Function Unit Statous:

Time	*Name*	*Busy*	*Op*	des *Fi*	*S1 Fj*	*S2 Fk*	*RS Qj*	*RS Qk*	*Rj*	*Rk*
	Intege	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Clock cycle counter

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Busy：用于描述功能部件硬件资源的占用情况
Op：说明运行时所用的是哪一个功能部件
Fi：目标寄存器
Fj, Fk：源操作数或寄存器
Qj, Qk：是否有某个功能单元占用源寄存器Fj或Fk。
Rj, Rk：用于标识源寄存器是否准备好。

Tip：
- 多说无益，自己须亲自推过
- 截取lab6一段代码，寄存器赋值顺序应该是91829，自己推一下

```
div x8, x7, x2
mul x9, x4, x5
mul x9, x8, x2
addi x2, x0, 4
jalr x1,0(x0)
```

第3章：指令级并行性（Scoreboard reaname）

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec start	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4	5
LD	F2	45+	R3	2	3	4	5	6
MULTD	F0	F2	F4	3	7	8	17	18
SUBD	F8	F6	F2	4	7	8	9	10
DIVD	F10	F0	F6	5	19			
ADDD	F6	F8	F2	11	12	13	14	15

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Int1	No								
	Int2	No								
	Mult1	No								
	Add	No								
40	Divide	Yes	Divd	P40	P36	P32	Mult1		NO	NO

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
19	FU	P36	P34	P4	P42	P38	P40	P12		P30

Tip：
- 多说无益，自己须亲自推过
- 区别在于硬件寄存器多，可以通过映射不同寄存器解决WAR, WAW问题

第3章：指令级并行性及其开发方法（Tomasulo）

Instruction status:

Instruction		j	k	Issue	Exec Start	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	2	3	4	5
MULTD	F0	F2	F4	3	6	15	16
SUBD	F8	F6	F2	4	6	8	9
DIVD	F10	F0	F6	5	17	56	57
ADDD	F6	F8	F2	6	10	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

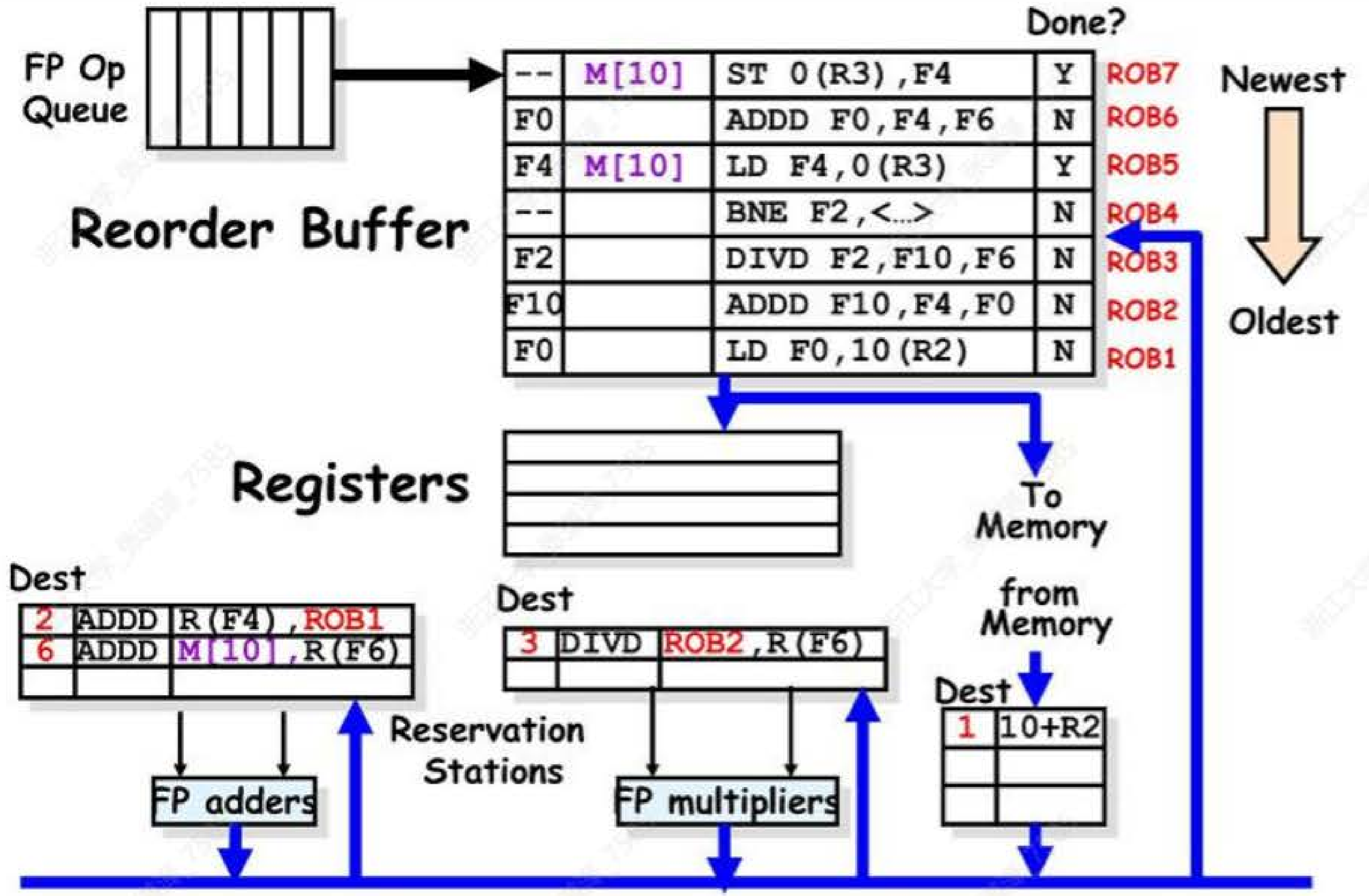
Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
57	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

Tip：
- 多说无益，自己须亲自推过
- 注意和scoreboard的区别，有保留站保存真实寄存器值，可以硬件解决WAW, WAR

Once again: In-order issue, out-of-order execution and out-of-order completion.

第3章：指令级并行性及其开发方法（投机执行）



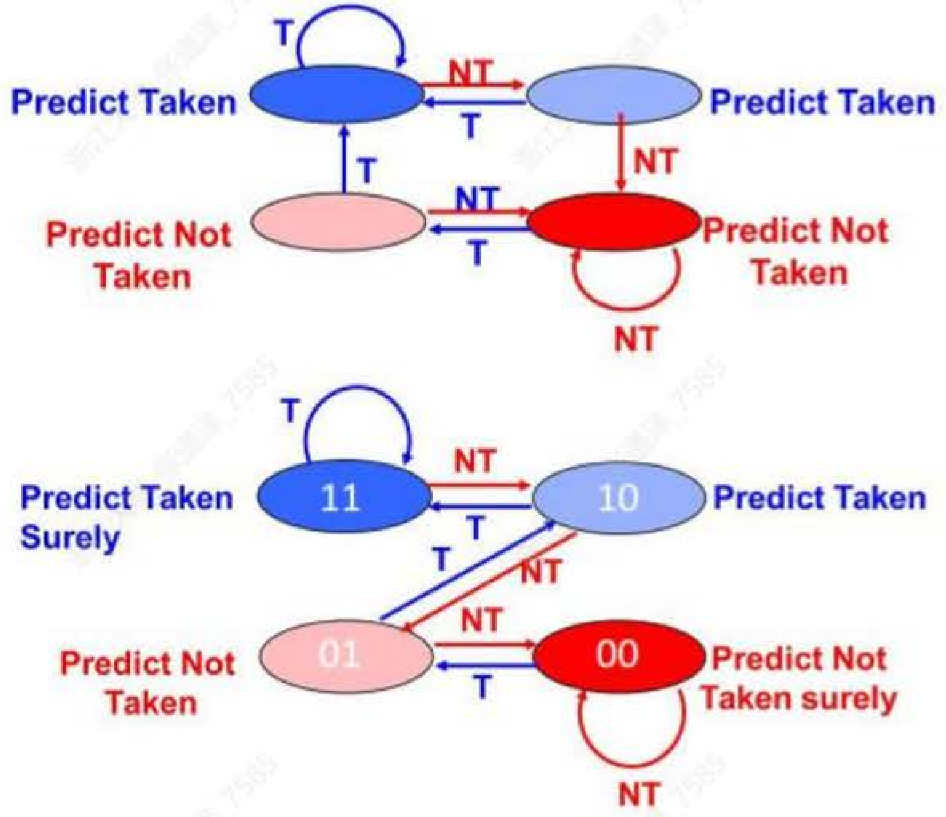
FP Op Queue
Reorder Buffer
Done?
-- | M[10] | ST 0(R3),F4 | Y | ROB7
F0 | | ADDD F0,F4,F6 | N | ROB6
F4 | M[10] | LD F4,0(R3) | Y | ROB5
-- | | BNE F2,<…> | N | ROB4
F2 | | DIVD F2,F10,F6 | N | ROB3
F10 | | ADDD F10,F4,F0 | N | ROB2
F0 | | LD F0,10(R2) | N | ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD M[10],R(F6)
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers

Tip：

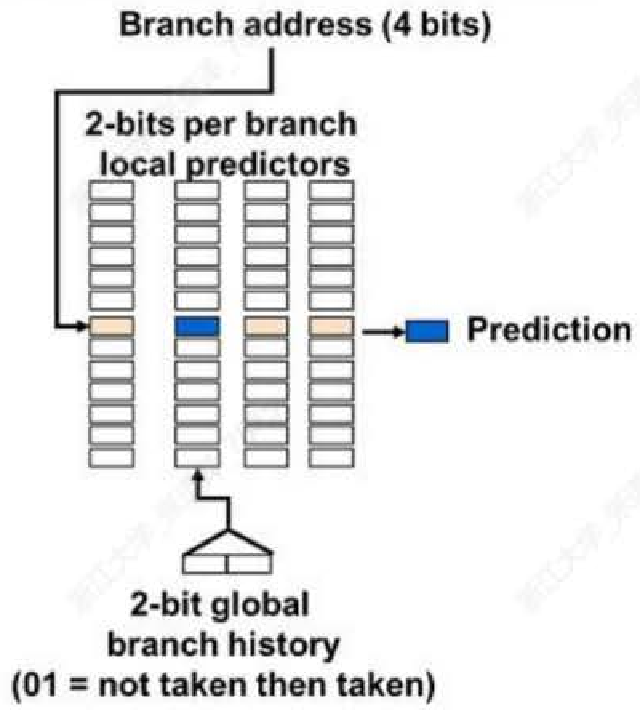
- 多说无益，自己须亲自推过
- 比Tomasulo（顺序发射乱序执行）多了一个顺序提交，设置ROB记录每一条指令的系信息

第3章：指令级并行性及其开发方法（动态预测）

2-bit Branch-Prediction Buffer



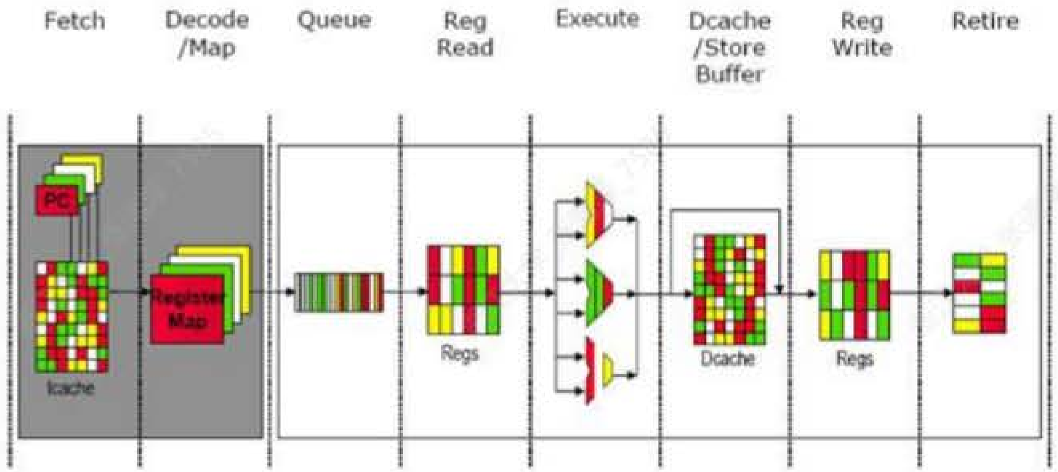
Correlating Branch Predictors


Branch address (4 bits)
2-bits per branch local predictors
Prediction
2-bit global branch history
(01 = not taken then taken)

(m,n)预测器的含义是观察该跳转前m条跳转语句，选择一个n位的对应预测器。（1,1）预测器自己结合PPT推一遍

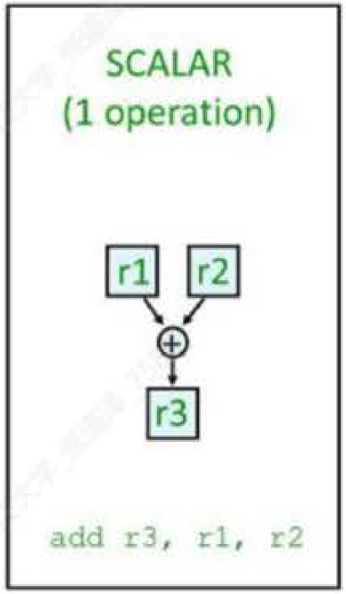
第3章：指令级并行性及其开发方法（其他）

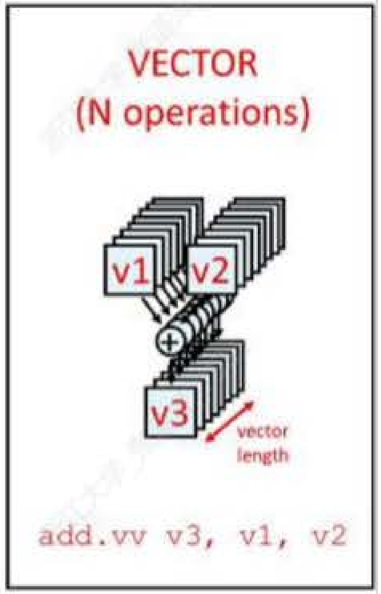
- 多发射
  - 之前所有方法每个周期只能发射1条指令，那么反过来每个指令至少需要1个周期，多发射每个周期能够发射多条指令，使得CPI可以小于1
- 循环展开
- 多线程



第4章：数据级并行GPU, vector, SIMD

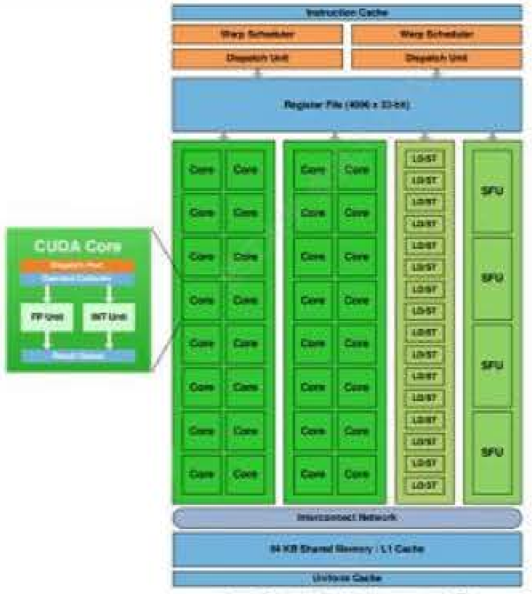
- SIMD三种变体
  - Vector，向量处理，以vector为基本粒度
  - SIMD，AVX指令集，支持多数据并行处理
  - GPU, SIMT异构体系
- Tip
  - 一些概念清楚就好Strip Mining 什么的，可能考一些计算周期的，分析方式和之前流水线分析冲突类似。重要的是计算题一定要写过程!





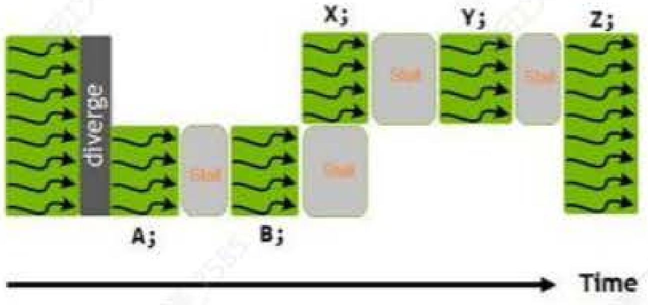
第4章：数据级并行GPU,vector,SIMD （补充）

- GPU架构 (以NVIDIA Fermi为例)
  - 物理架构
    - 每个GPU有多个SM
    - 每个SM内部有多个计算单元core，他们执行任务接受SM调度器的调度，以warp为粒度
- 存储架构
  - 每一个SM独有一个L1 cache/SMEM
  - 在global memory和SM之间有L2 cache



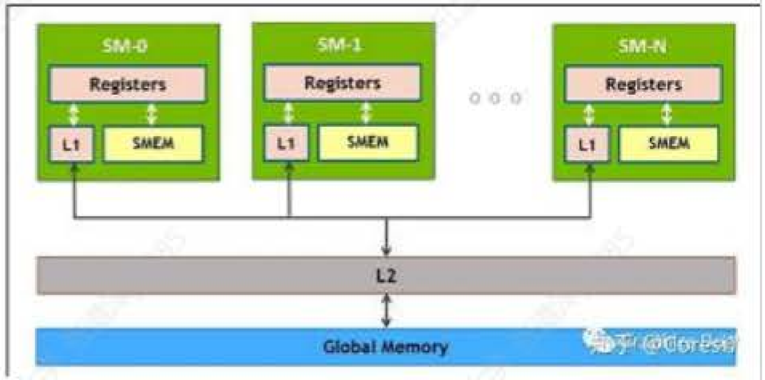
Fermi Streaming Multiprocessor (SM)

```
if (tid<4){
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

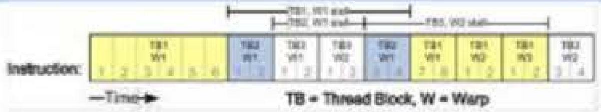


```
if (x == 0)
{
    wait(a);
    y--;
}
y++;
signal(b);
```

死锁代码



第4章：数据级并行GPU,vector,SIMD （补充）



- GPU硬件层次结构
  - SP: streaming processor，一个计算核心Core，可以理解成1个小CPU，对应1个线程的计算
  - SM: streaming multiprocessor，包含很多SP。SM的数量几十几百都可能，看架构。任务接受SM调度器调度，基本单位是warp（32线程）。运行时一个SM可以同时容纳多个warp
  - Device：指GPU板卡，可以有多个SM
- GPU软件层次结构
  - Thread：一个线程，执行的最小粒度，映射到一个core中
  - Block：包含很多个thread，block内共享Inst. code（SIMT），在执行时block会以warp的形式打包分发给空闲的SM上。通常block内线程数是warpSize(32)的整数倍，以免线程束空闲
  - Grid：包含多个block，用于更高层抽象。通常包含block的数量选择SM数量的整数倍，便于负载均衡

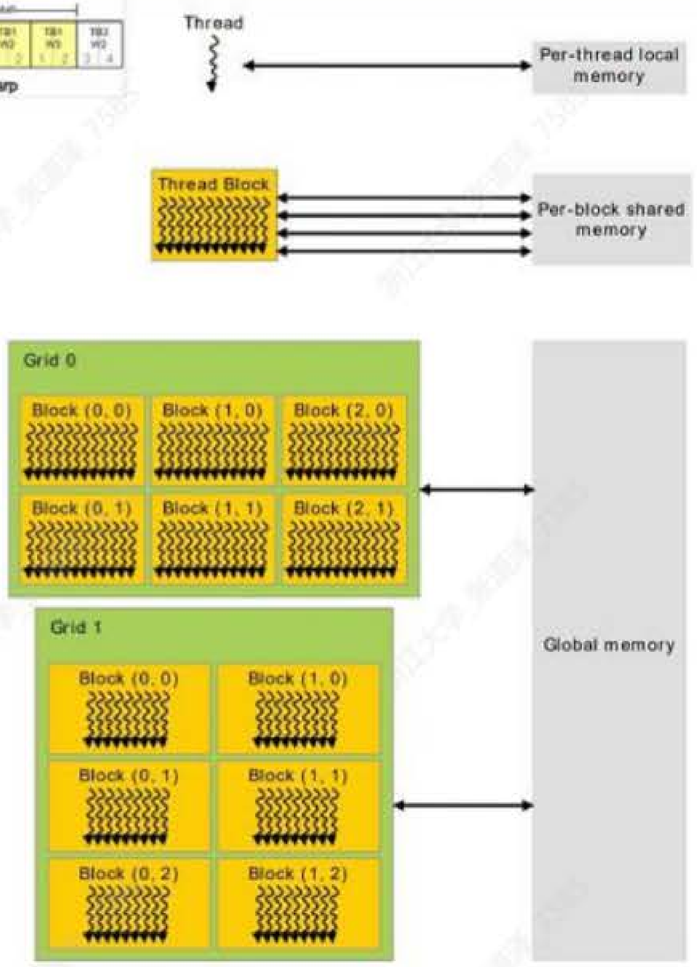
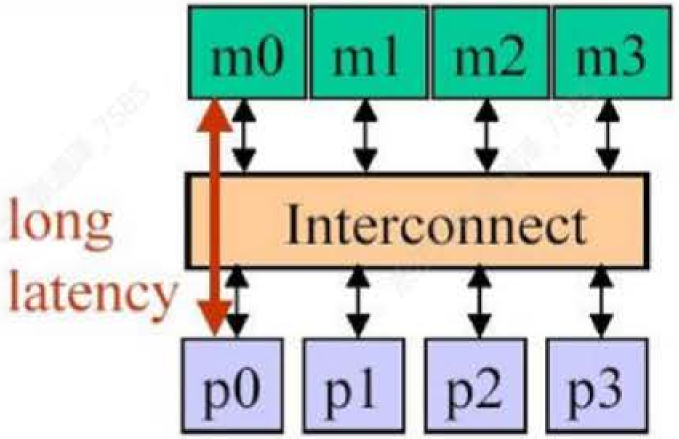
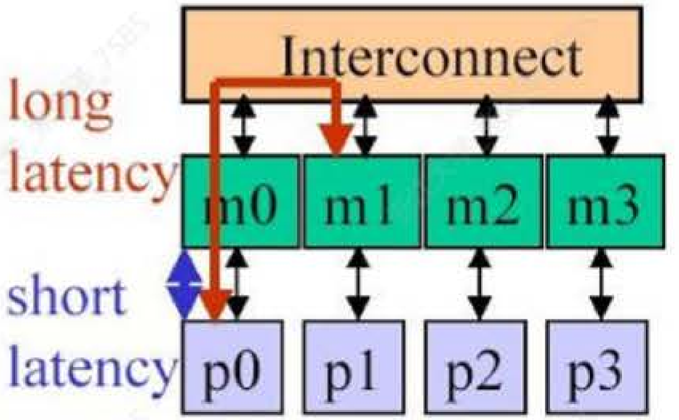


Figure 7 Memory Hierarchy

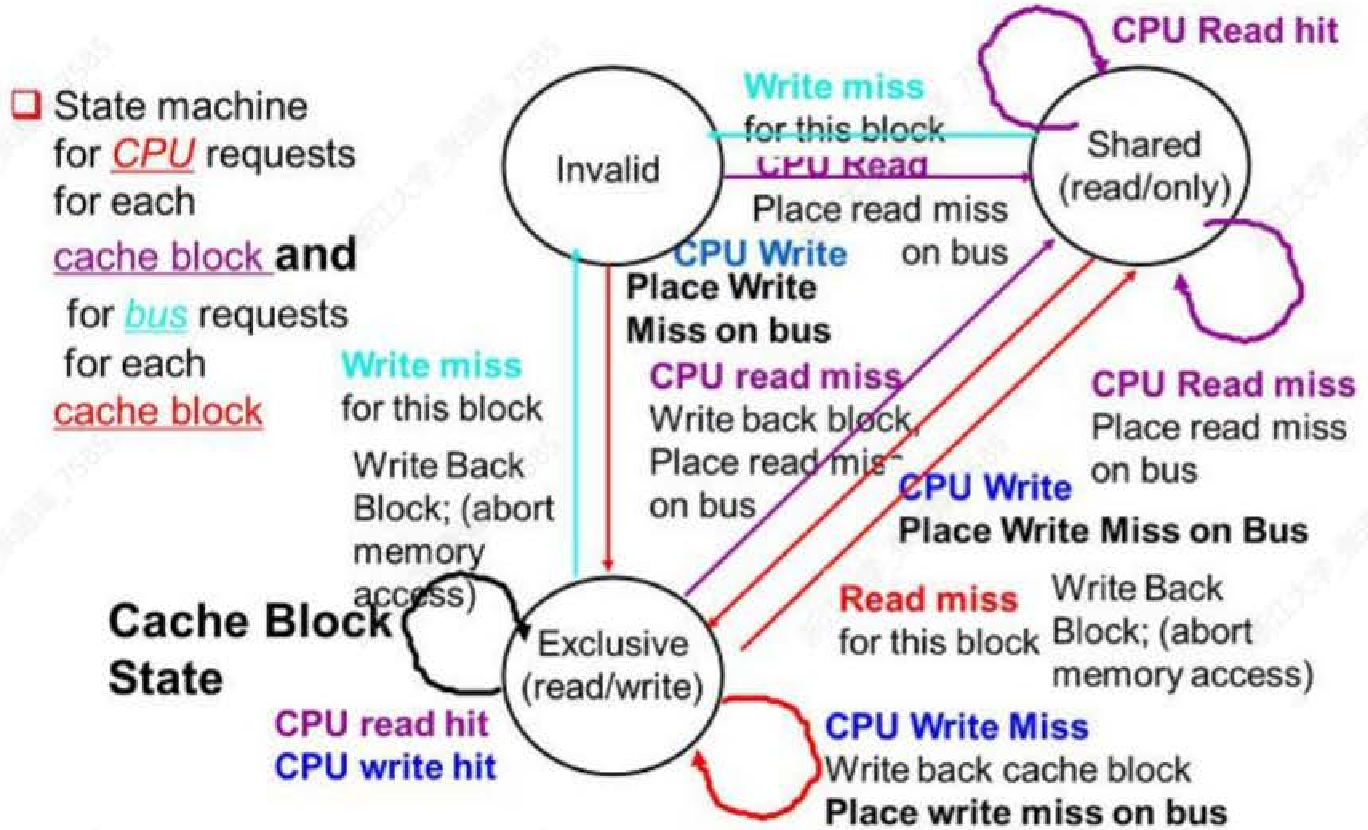
第5章 线程级并行

- 多线程
  - UMA/NUMA shared/distribute
  - 提供线程级并行同时会产生通信开销，统一数据多份副本，造成缓存一致性问题与存储连续性问题，需要同步等
- 缓存一致性
  - 同一个数据多个副本，cache coherence通过一些手段（snoopy directory）保证同一个数据的信息在多个副本中正确传递
- 存储连续性
  - 存储相关指令执行对单核可以是乱序的，但多核协调的程序正确性需要对存储相关指令进行约束





第5章 线程级并行（缓存一致性）



Snooping

第5章 线程级并行（缓存一致性）

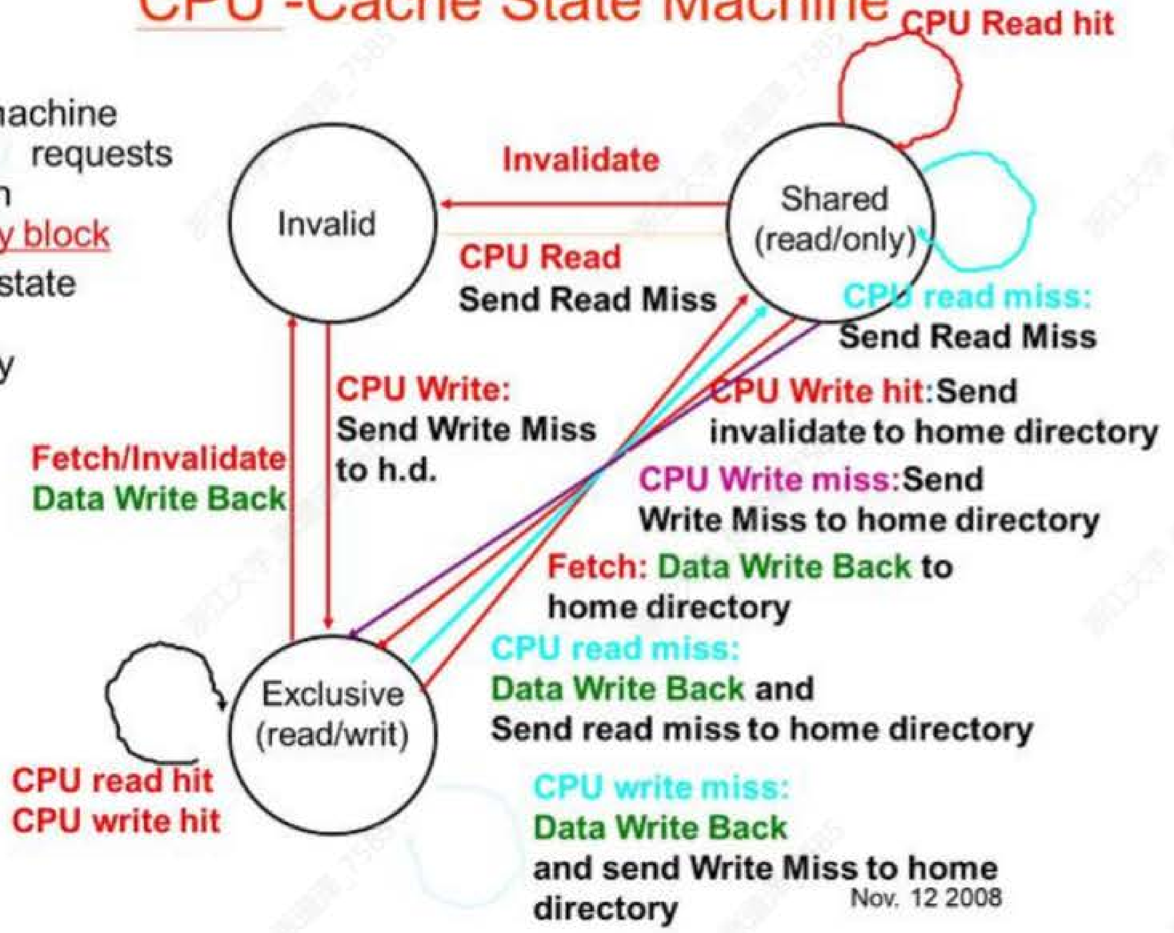
Message type	***Source***	***Destination***	***Msg Content***
Read miss	Local cache	Home directory	P, A
➢ *Processor P reads data at address A; make P a read sharer and arrange to send data back*			
Write miss	Local cache	Home directory	P, A
➢ *Processor P has a write miss at address A; make P the exclusive owner and arrange to send data back*			
Invalidate	Local cache	Home directory	A
➢ *Request to send invalidates to all remote caches that are caching the block at address A*			
Invalidate	Home directory	Remote caches	A
➢ *Invalidate a shared copy at address A.*			
Fetch	Home directory	Remote cache	A
➢ *Fetch the block at address A and send it to its home directory*			
Fetch/Invalidate	Home directory	Remote cache	A
➢ *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	Home directory	Local cache	Data
➢ *Return a data value from the home memory (read miss response)*			
Data write-back	Remote cache	Home directory	A, Data
➢ *Write-back a data value for address A (invalidate response)*			

Directory

第5章 线程级并行（缓存一致性）

CPU -Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory

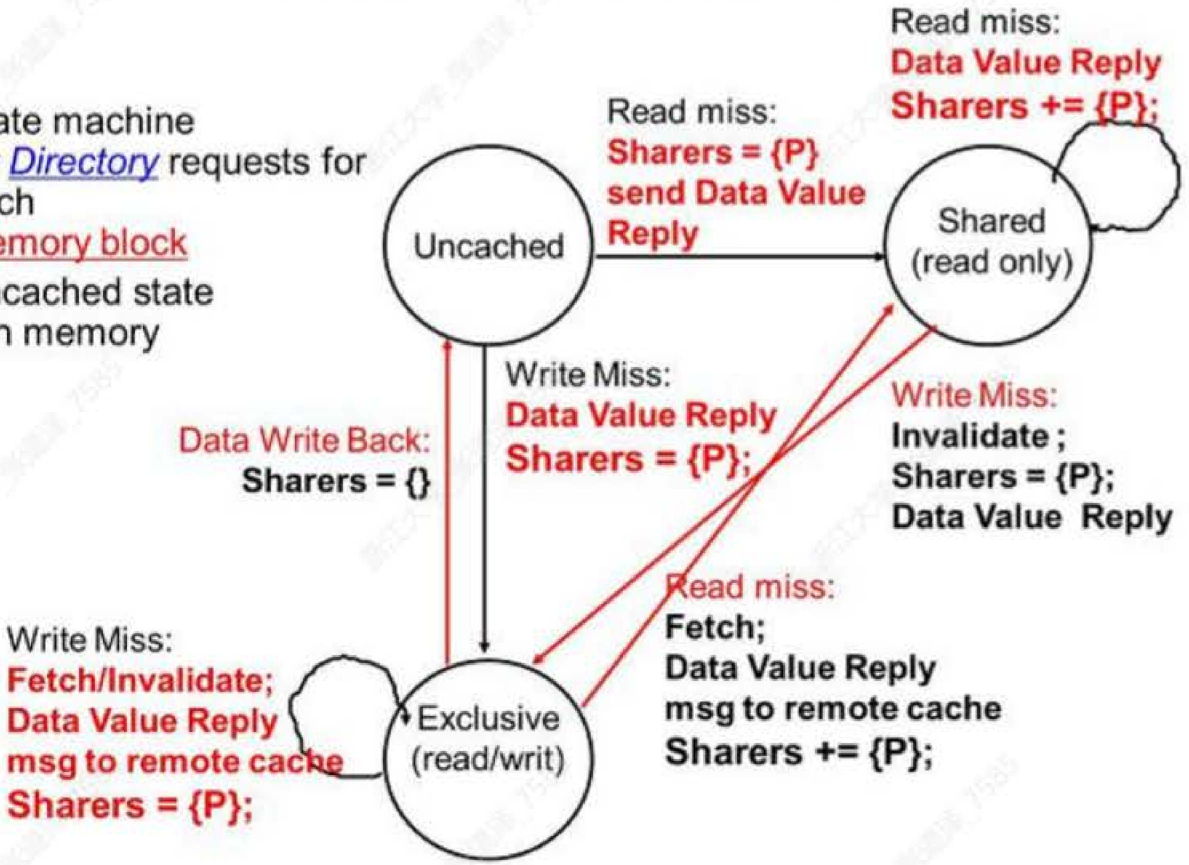


Directory

第5章 线程级并行（缓存一致性）

Directory State Machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



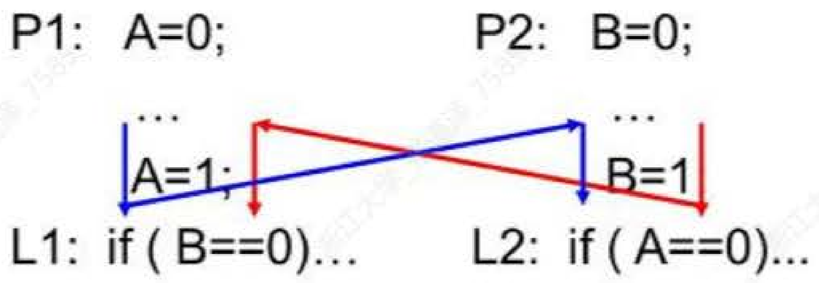
Directory

第5章 线程级并行（同步）

- Tip
  - 了解同步相关的硬件指令含义
  - 计算题抄在A4纸上参考

第5章 线程级并行（访存一致性）

- Memory accesses executed by each processor were kept in order
- Memory accesses among different processors were interleaved.

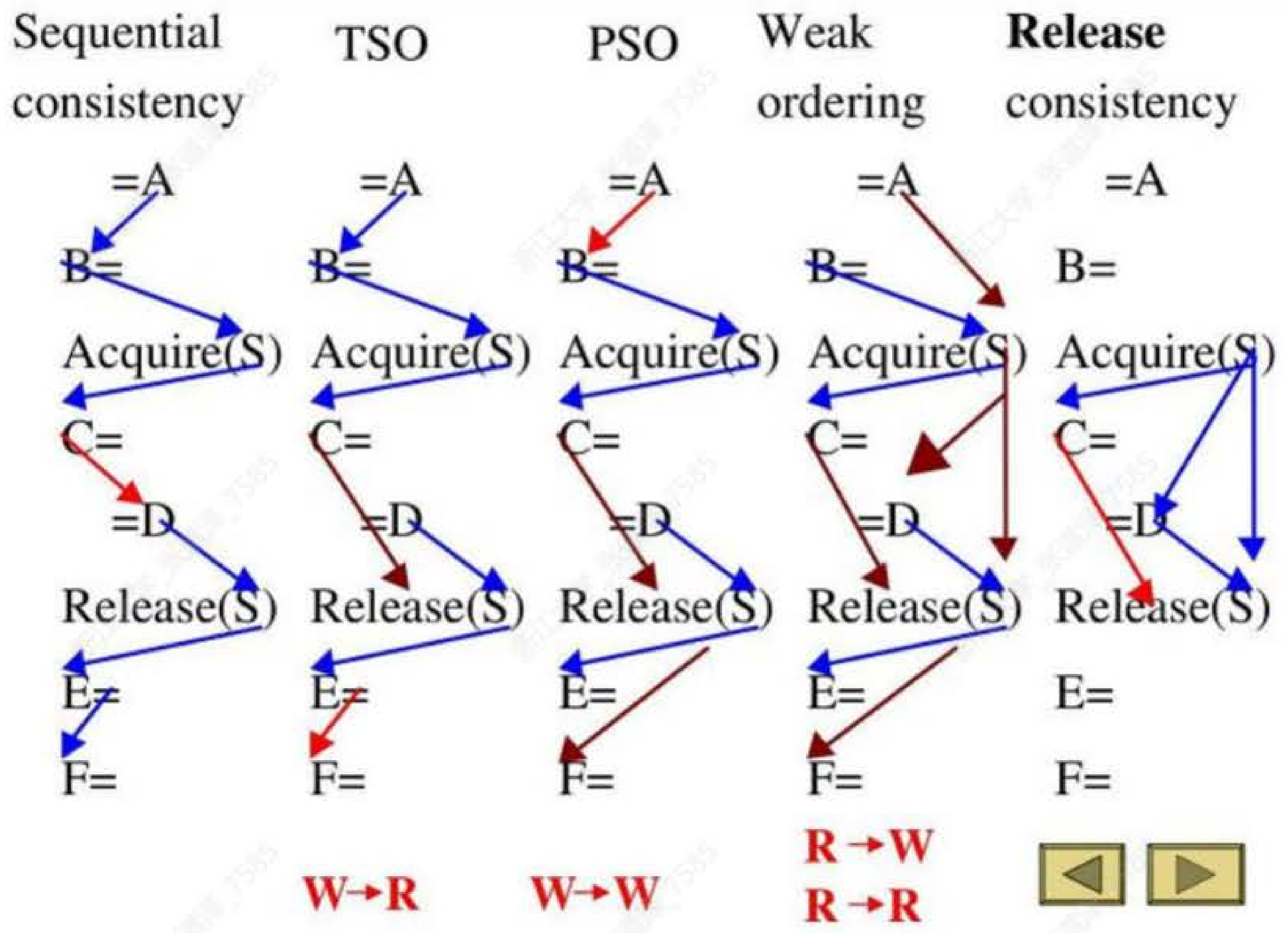


False True/false

True/false False

No way to see True Ture !

第5章 线程级并行（访存一致性）



Sequential consistency
TSO
PSO
Weak ordering
Release consistency
=A
B=
Acquire(S)
C=
=D
Release(S)
E=
F=
W→R
W→W
R→W
R→R