
ISA

How to Improve the CPU

- Reduce the number of instructions
    - Make instructions that do more(CSIC)
    - Use better compiler
- Use less cycles to perform the instruction
    - Simpler instructions(RISC)
    - Use multiple units/ALUs/cores in parallel
- Increase the clock frequency
    - Find a newer technology to manufacture
    - Redesign time critical components
    - Adopt multi-cycle

ISA Classification Basis

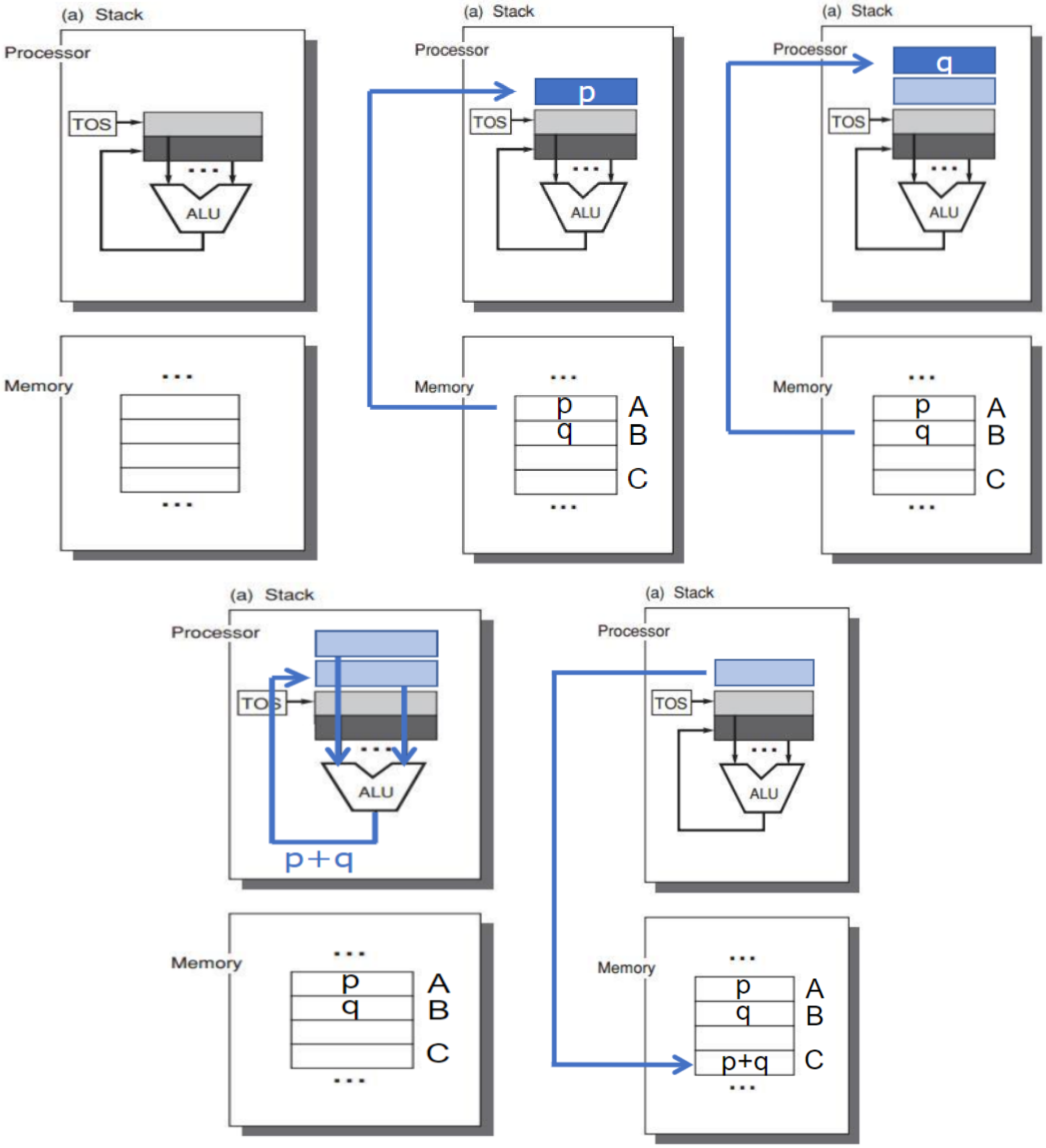
ISA Classes

- 按照CPU内部的存储分类，ISA可以分为三种
    1. 基于栈（栈结构 | Stack architecture)
    2. 基于累加器（累加器结构 | Accumulator architecture)
    3. 基于寄存器（通用寄存器 | General-purpose register architecture | GPR)

Stack Architecture

- Implicit Operands——On the Top of the Stack(TOS)
- First operand removed from second op replaced by the result
- C = A + B(memory locations)
    - Push A

  - Push B
  - Add
  - Pop C



Accumulator Architecture

- One implicit operand: the accumulator

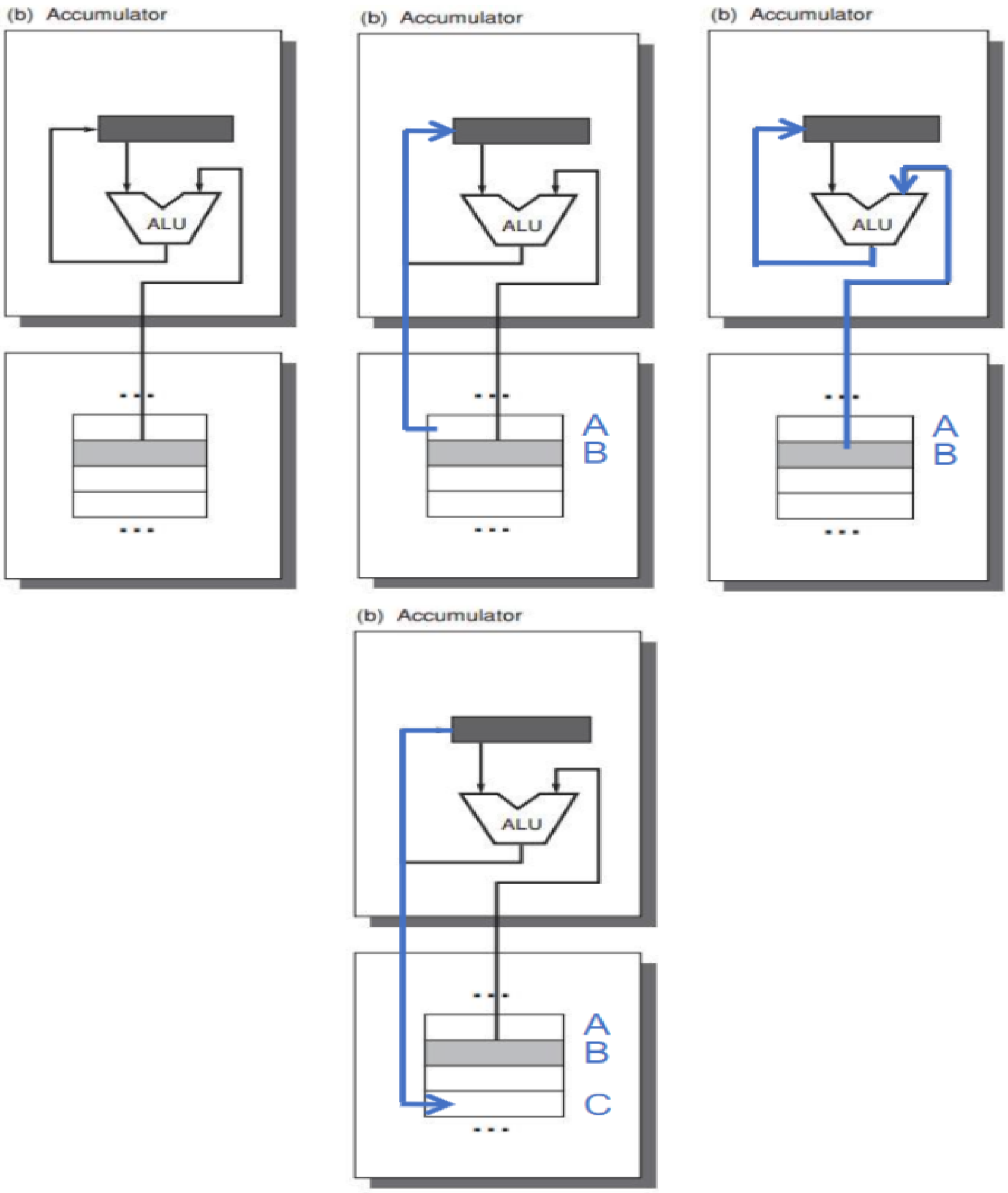
  one explicit operand: mem location

- Accumulator is both an implicit input operand and a result
- C = A + B

  Load A

  Add B

  Store C



GPR Arthitecture

- Only explicit operands
    - registers
    - memory locations
- Operand access:
    - direct memory access
    - loaded into temporary storage first
- Two Classes:
    - Register-memory architecture

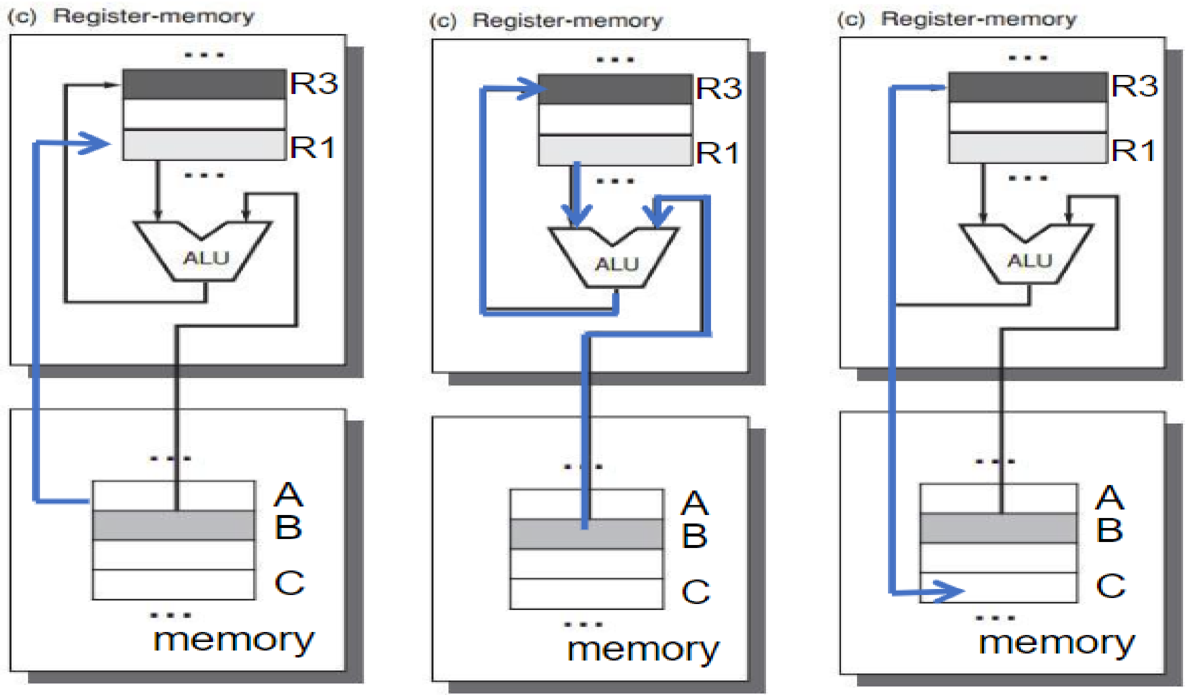
```
any instruction can access memory
```

- Load-store architecture

```
only load and store instructions can access memory
```

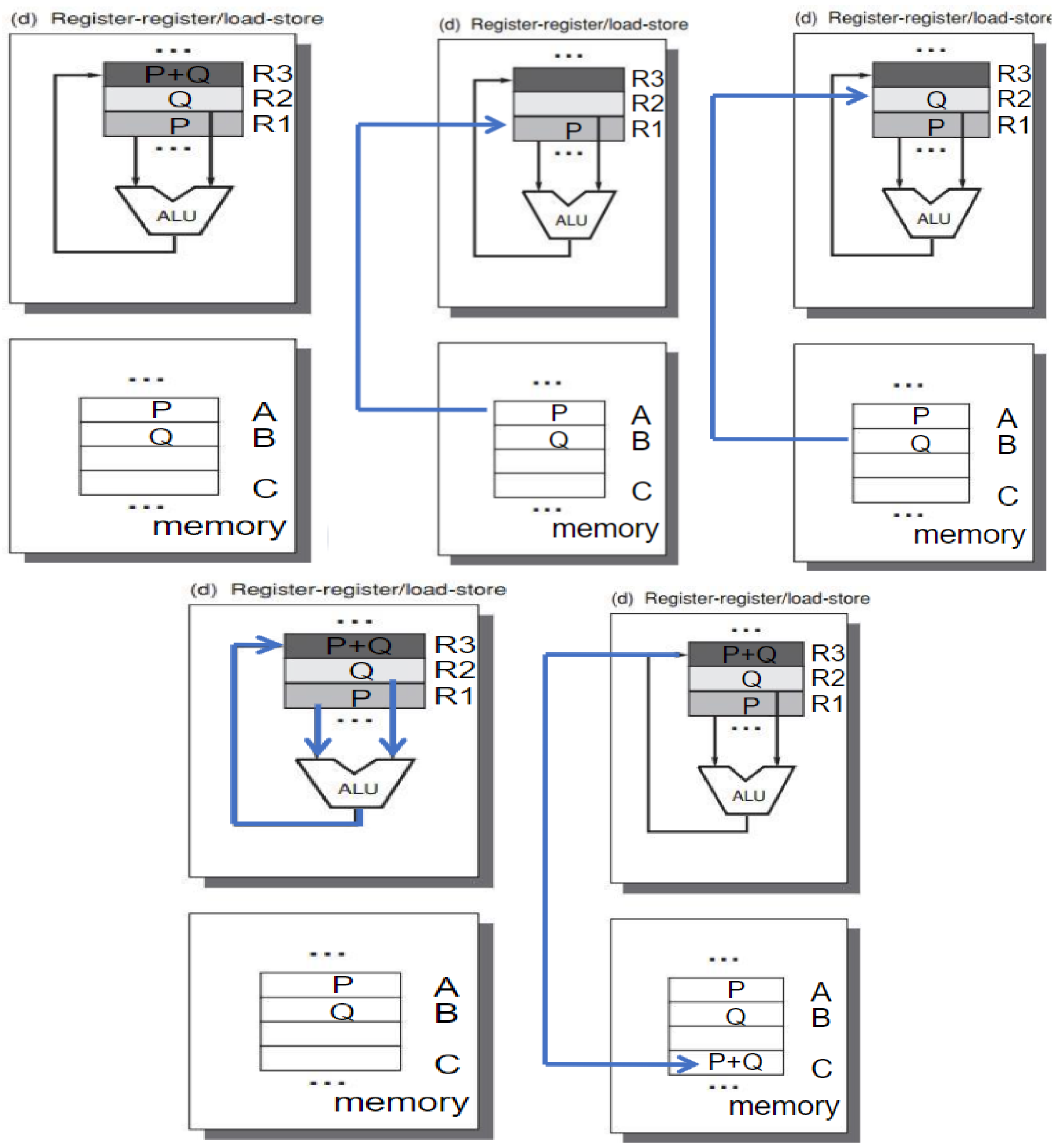
GPR: Register-Memory Arch

- Register-memory architecture(any instruction can access memory)
- C = A + B
  - Load R1, A
  - Add R3, R1, B
  - Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture(only load and store instructions can access memory)
- C = A + B
  - Load R1, A
  - Load R2, B
  - Add R3, R1, R2
  - Store R3, C


(d) Register-register/load-store
P+Q
Q
P
R3
R2
R1
ALU
P
Q
A
B
C
memory
(d) Register-register/load-store
R3
R2
R1
P
ALU
P
Q
A
B
C
memory
(d) Register-register/load-store
Q
P
R3
R2
R1
ALU
P
Q
A
B
C
memory
(d) Register-register/load-store
P+Q
Q
P
R3
R2
R1
ALU
P
Q
A
B
C
memory
(d) Register-register/load-store
P+Q
Q
P
R3
R2
R1
ALU
P
Q
P+Q
A
B
C
memory

Practice

$$D = A \times B - (A + C \times B)$$

Stack

```
Instruction set:
  • add, sub, mul, div, ...
  • push A, pop A
```

- Answer
  1. push A
  2. push B

3. mul
4. push A
5. push C
6. push B
7. mul
8. add
9. sub
10. pop D

Accumulator

> Instruction set:
>
> - add A, sub A, mult A, div A, ...
> - load A, store A

- Answer:
  1. load C
  2. mult B
  3. add A
  4. store D
  5. load A
  6. mult B
  7. sub D
  8. store D

Memory-Memory

> Instruction set:
>
> - 3 operands: add A, B, C; sub A, B, C; mul A, B, C
> - 2 operands:add A, B; sub A, B; mul A, B; mov A, B

- Answer:
  - 3 operands:
    - mul D, A, B
    - mul E, C, B
    - add E, A, E
    - sub D, D, E
  - 2 operands:
    - mov D, A
    - mul D, B
    - mov E, C
    - mul E, B
    - add E, A

        - sub D, E

Register-Memory

> Instruction set:
>
> - add R1, A; sub R1, A; mul R1, B
> - load R1, A; store R1, A

- Answer:
    - load R1, A
    - mul R1, B
    - load R2, C
    - mul R2, B
    - add R2, A
    - store R2, D
    - sub R1, D
    - store R1, D

Load-Store

> Instruction set:
>
> - add R1, R2, R3; sub R1, R2, R3; mul R1, R2, R3
> - load R1, &A; store R1, &A

- Answer:
    - load R1, &A
    - load R2, &B
    - load R3, &C
    - mul R3, R3, R2 // R3 = B*C
    - add R3, R3, R1 // R3 = A+B*C
    - mul R1, R1, R2 //R1 = A*B
    - sub R1, R1, R3
    - store R1, D

RISC-V ISA

Formats of Instruction

> 不需要记住具体哪位对应什么，需要记住R型指令是什么，I型是什么……

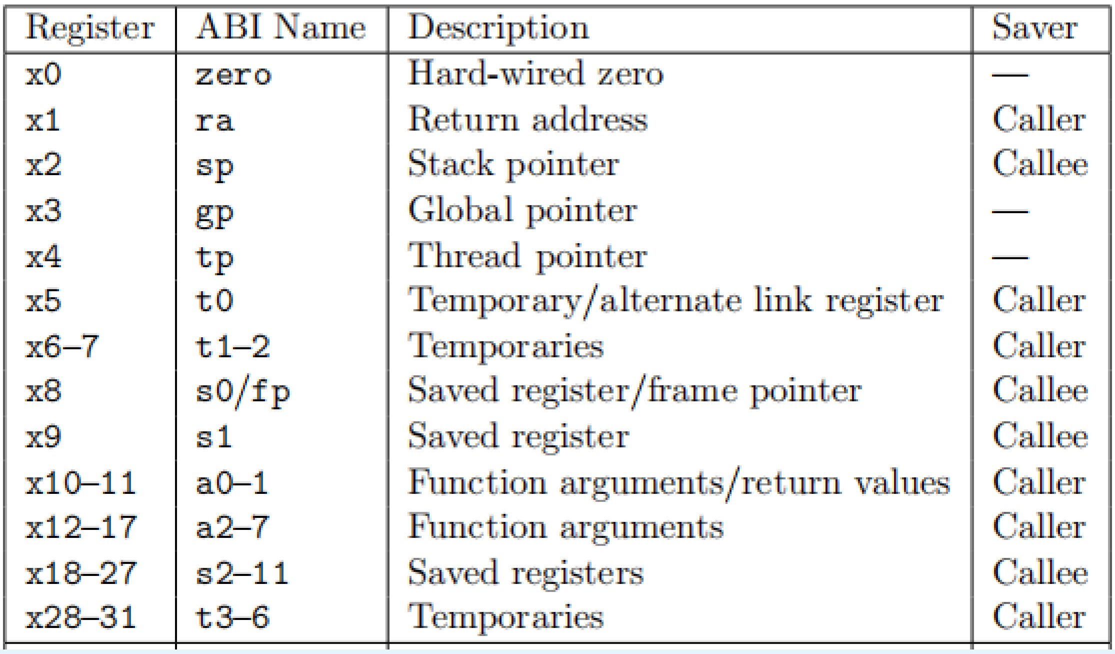
RISC-V Address Mode

四种寻址方式

1. 立即数寻址 | Immediate addressing
2. 寄存器寻址 | Register addressing
3. 基寻址 | Base addressing
4. PC-relative addressing

Register Operands

1. Arithmetic instructions use register operands
2. RISC-V has a 32×32-bit register file
    1. Use for frequently accessed data
    2. Numbered 0 to 31
    3. 32-bit data called a "word"
3. Assemble names
    1. x0: constant 0
    2. x1: link register
    3. x2: stack pointer
    4. x3: global pointer
    5. x4: thread pointer
    6. x5-x7, x28-x31: temporay
    7. x8-x9, x18-x27: save
    8. x10-x17: parameter / result



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

- Caller在调用新函数时不会保存，会被重写（返回新值）
- Callee在调用时会保存不变，无法返回新值

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into register
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- RISC-V is Little Endian
  - Least-significant byte at least address
  - c.f.Big Endian: most-significant byte at least adddress of a word

> Example:
>
> - C code:
>   - `g = h + A[8];`
>   - g in x8, h in x9, base address of A in x18
> - Compiled RISC-V code:
>   - Index 8 requires offset of 32
>   - 4 bytes per word
>   - `lw x5, 32(x18) #偏移32是因为A[8]是以A的地址偏移8*4=32`
>   - `add x8, x9, x5`

Register vs. Memory

- Register are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important

The Four ISA Design Principles

1. Simplicity favors regularity | 简单明了有助于一致性
2. Smaller is faster | 越小越快
3. Make the common case fast | 加快经常性事件的速度

> 例如x0是硬件0，无法改变

4. Good design demands good compromises | 好的设计需要好的妥协

Pipelining

Different modes of execution

- Sequential execution
  - Advantages: Simple control, saving equipment.
- Single/Twice overlapping execution
  - Advantages: High-usage of functional unit, the utilization rate of functional unit is improved obviously.
  - Disadvantages:
    - Much more hardware was needed.
    - Single: the control process became complicated.
    - Twice:
      - Separate fetch, decode, and execution components are required.
      - Conflict in access memory
        - Instruction memory & data memory
        - Instruction cache & data cache (same memory): Hardware structure
        - Adding instruction buffer between memory and instruction decode unit.

> 顺序执行和重叠执行在时间上的差异和计算

What is pipelining?

- Pipelining: 一条指令的进程被分成m(m>2)个时间相同的子进程，且m条相邻指令的进程在同一时间内交错重叠。
- Pipelining可以看作重叠执行(overlapping execution)的扩展;
- 流水线中的每个子进程及其功能组件叫做阶段(stages or segments of the pipeling)
- 阶段的数量叫做流水线的深度(depth)

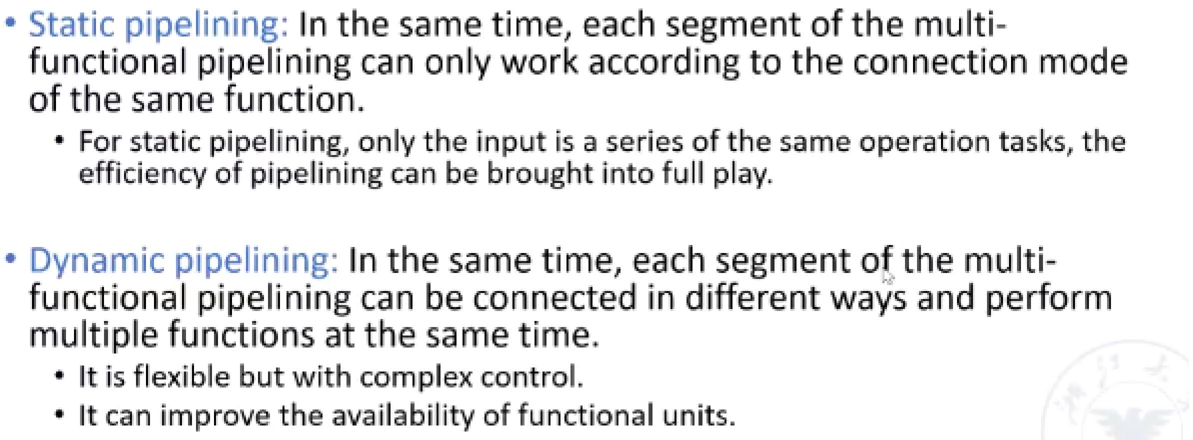
Characteristics of pipelining

- The pipelining divides a process into several sub processes, each of which is implemented by a special functional unit.
- The time of each stage in the pipelining should be equal as much as possible,otherwise the pipelining will be blocked and cut off. A longest stage will become the bottleneck of the pipelining.

- Every functional part of the pipelining must have a buffer register (latch), which is called pipelining register.
    - Function: transfer data between two adjacent stages to ensure the data to be used later, and separate the processing work of each stage from each other.
- Pipelining technology is suitable for a large number of repetitive sequential processes. Only when tasks are continuously provided at the input, te efficiency of pipelining can be brought into full play.
- The pipelining needs the pass time and the empty time
    - Pass time: the time for the first task from beginning(entering the pipelining) to ending.
    - Empty time: the time for the last task entering the pipelining to have the result.

Classes of pipelining

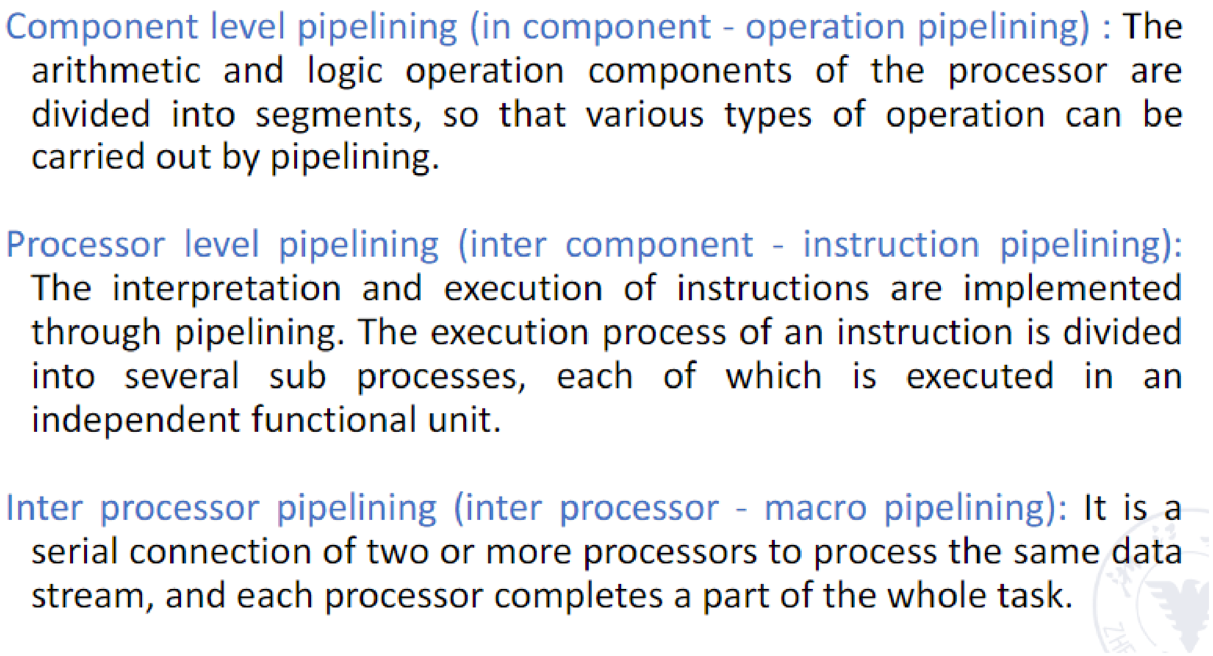
- 第一类分类方式:
    - Single function pipelining | 单功能: only one fixed function pipelining.
    - Multi function pipelining | 多功能: each section of the pipelining can be connected differently for several different functions.
        - Static pipelining: 静态在同一的时间段内，只能按同一功能连接（如：必须先把加法做完，才可以做乘法）
            - For static pipelining, only the input is a series of teh same operation tasks, the efficiency of pipelining can be brought into full play.
        - Dynamic pipelining: 动态在同一时间段内，不同功能可以连接（如：在做加法的时候可以插入乘法）



- Static pipelining: In the same time, each segment of the multi-functional pipelining can only work according to the connection mode of the same function.
    - For static pipelining, only the input is a series of the same operation tasks, the efficiency of pipelining can be brought into full play.
- Dynamic pipelining: In the same time, each segment of the multi-functional pipelining can be connected in different ways and perform multiple functions at the same time.
    - It is flexible but with complex control.
    - It can improve the availability of functional units.

- 第二类分类方式:
    - Component level pipelining(in component-operation pipelining:
        - 根据不同的操作，加法和乘法可以同时做，实现operation中的pipeline
    - Processor level pipelining(inter component-instructino pipelining):
        - 相当于本学期做的CPU

    - Inter processor pipelining(inter processor-macro pipelining):
        - 将不同的指令进行集合，形成一个宏去实现某一个特定的功能，宏和宏之间实现pipeline



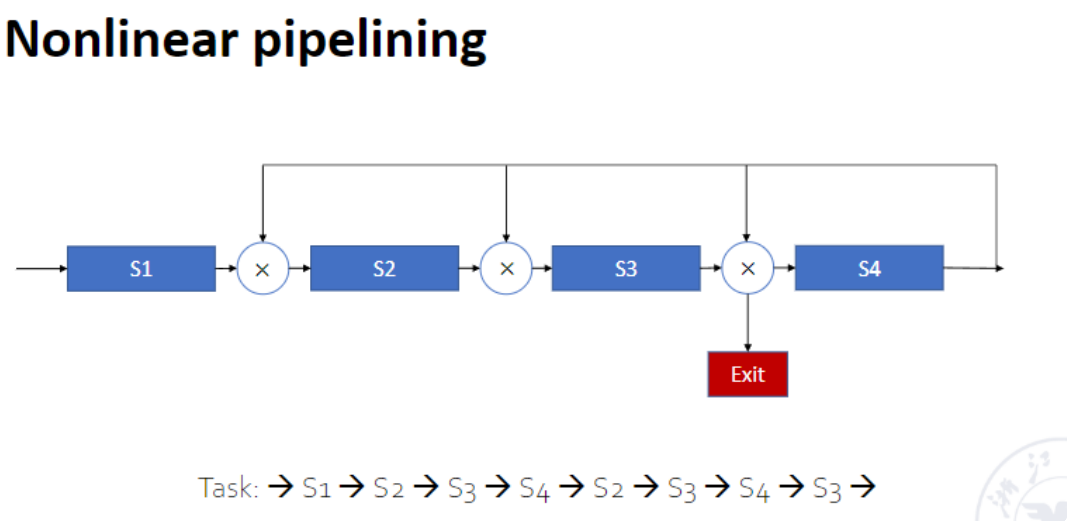
Component level pipelining (in component - operation pipelining) : The arithmetic and logic operation components of the processor are divided into segments, so that various types of operation can be carried out by pipelining.

Processor level pipelining (inter component - instruction pipelining): The interpretation and execution of instructions are implemented through pipelining. The execution process of an instruction is divided into several sub processes, each of which is executed in an independent functional unit.

Inter processor pipelining (inter processor - macro pipelining): It is a serial connection of two or more processors to process the same data stream, and each processor completes a part of the whole task.

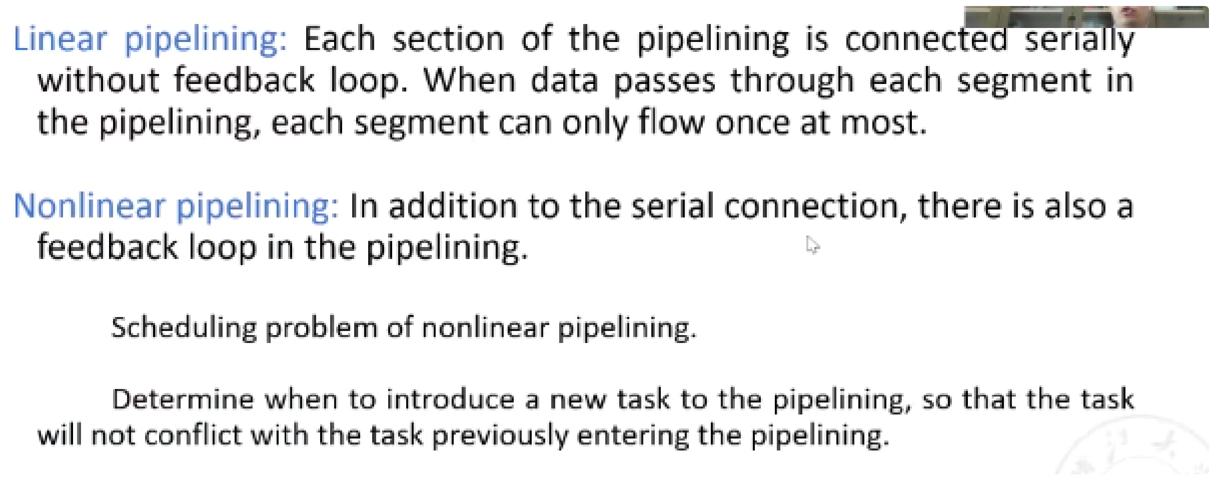
- 第三类分类方式：线性和非线性
    - Linear pipelining
    - Nonlinear pipelining
        - 存在一种基于表的调度，满足一定顺序才能实现某种功能

        非线性流水线调度问题

        - 

Nonlinear pipelining

Task: → S1 → S2 → S3 → S4 → S2 → S3 → S4 → S3 →



Linear pipelining: Each section of the pipelining is connected serially without feedback loop. When data passes through each segment in the pipelining, each segment can only flow once at most.

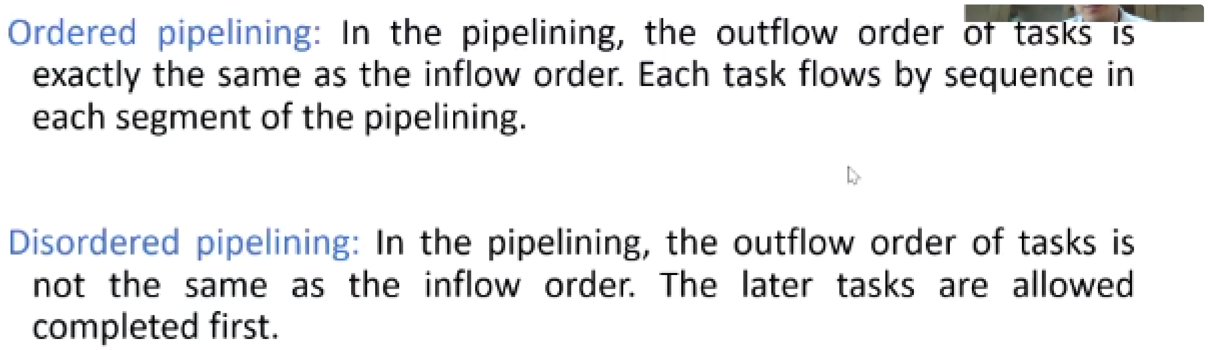
Nonlinear pipelining: In addition to the serial connection, there is also a feedback loop in the pipelining.

Scheduling problem of nonlinear pipelining.

Determine when to introduce a new task to the pipelining, so that the task will not conflict with the task previously entering the pipelining.

- 第四类分类
  - Ordered pipelining
  - Disordered pipelining

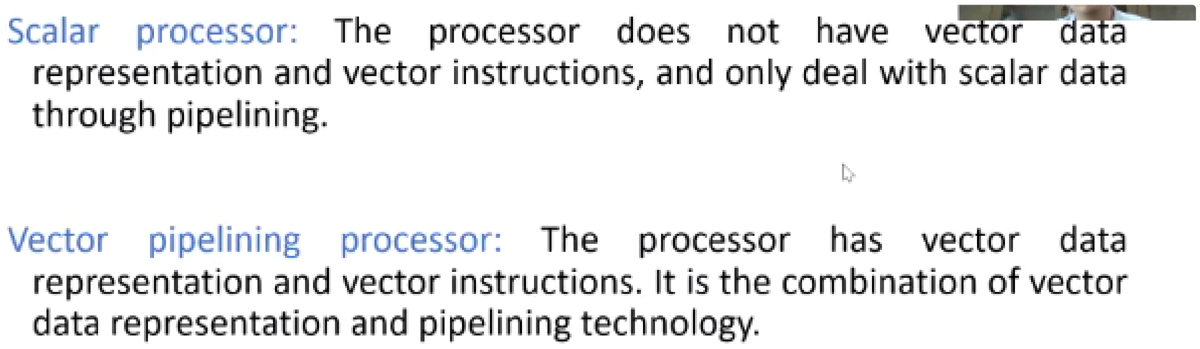
> Disordered类流水线和非线性都很复杂，存在调度问题



Ordered pipelining: In the pipelining, the outflow order of tasks is exactly the same as the inflow order. Each task flows by sequence in each segment of the pipelining.

Disordered pipelining: In the pipelining, the outflow order of tasks is not the same as the inflow order. The later tasks are allowed completed first.

- 第五类分类
  - Scalar processor
  - Vector pipelining processor



Scalar processor: The processor does not have vector data representation and vector instructions, and only deal with scalar data through pipelining.

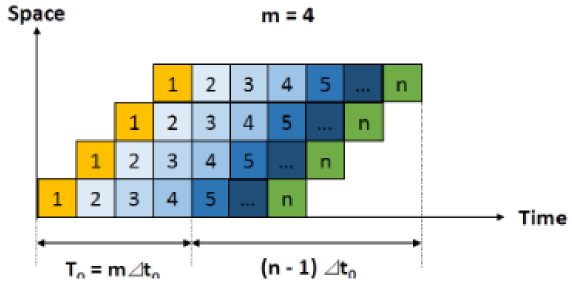
Vector pipelining processor: The processor has vector data representation and vector instructions. It is the combination of vector data representation and pipelining technology.

Pipeline Performance

Throughput(TP) | 吞吐量

- 单位时间内流过流水器的指令条数→$TP = \frac{n}{T}$，$TP < TP_{max}$

e.g.



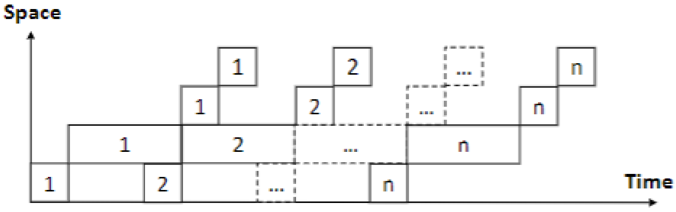
$$T = (m + n - 1) \times \triangle t_0$$

$$TP = n / (m + n - 1) \triangle t_0$$

$$TP_{max} = 1 / \triangle t_0$$

- 其中Δt_0为pipeline的时钟周期，即时间最长阶段的时间；
- m为阶段数
- n为指令条数

- 当时钟周期确定的时候，TP的理论上界其实就是确定的
  - The actual throughput of the pipeline is less than the maximum throughput, which is not only related to the time of each segment, but also related to m and n.
  - If $n \gg m$，$TP \approx TP_{max}$
- TP under Pratical Case
  - 实际情况下各阶段运行时间不同，最长的阶段叫做瓶颈阶段(bottleneck segment)



$$TP = \frac{n}{T}$$

$$TP = \frac{n}{\sum_{i=1}^{m} \Delta t_i + (n - 1) max(\Delta t_1, \Delta t_2, \dots, \Delta t_m)} = \frac{n}{(m-1)\Delta t + n\, max(\Delta t_1, \Delta t_2, \dots, \Delta t_m)}$$

$$TP_{max} = \frac{1}{max(\Delta t_1, \Delta t_2, \dots, \Delta t_m)}$$

$$\Downarrow$$

$$TP_{max} = \frac{1}{3\Delta t}$$

  - 解决bottleneck的方法：
    - Subdivision:
      - 将瓶颈阶段细分为可以流水线操作的更小阶段
    - Repetition:
      - 重叠执行不同指令的瓶颈阶段(意味着要增加硬件，来满足同时执行三个第二阶段)

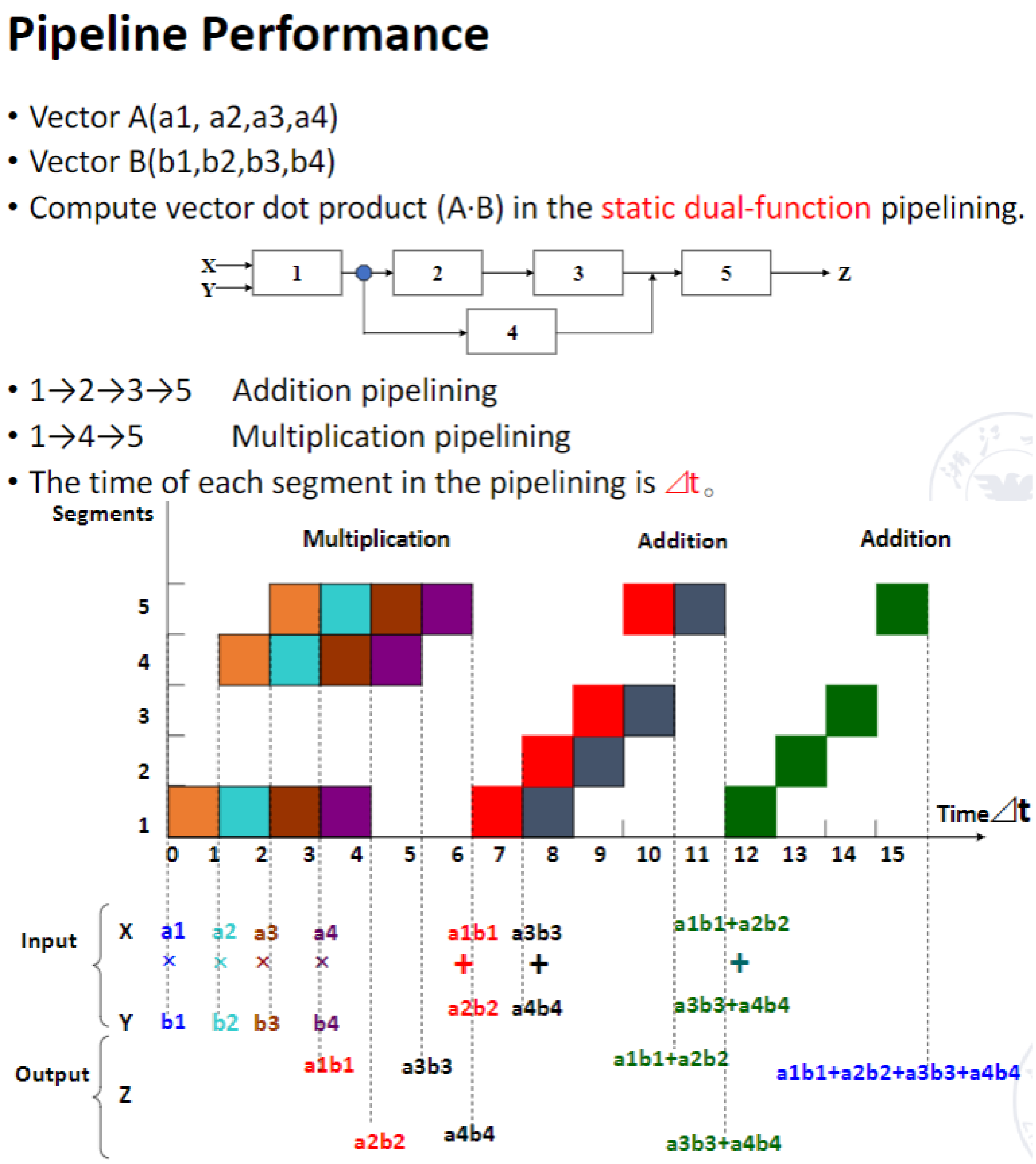
Speedup(Sp) | 加速比

- $Sp = \frac{Execution\ time_{non-pipelined}}{Execution\ Time_{pipelined}}$
- 以 `TP` 中的图为例，计算得到$Sp = \frac{n \cdot m \cdot \Delta t_0}{(m+n-1)\Delta t_0} = \frac{nm}{m+n-1}$
- 当$n \gg m$时，$Sp \approx m$

Efficiency(η) | 效率

- $\eta = \frac{Sp}{m}$（实际上使用的时空区域/总的时空区域≈使用了的面积/总面积）
- 当$n \gg m$时，$\eta = 1$

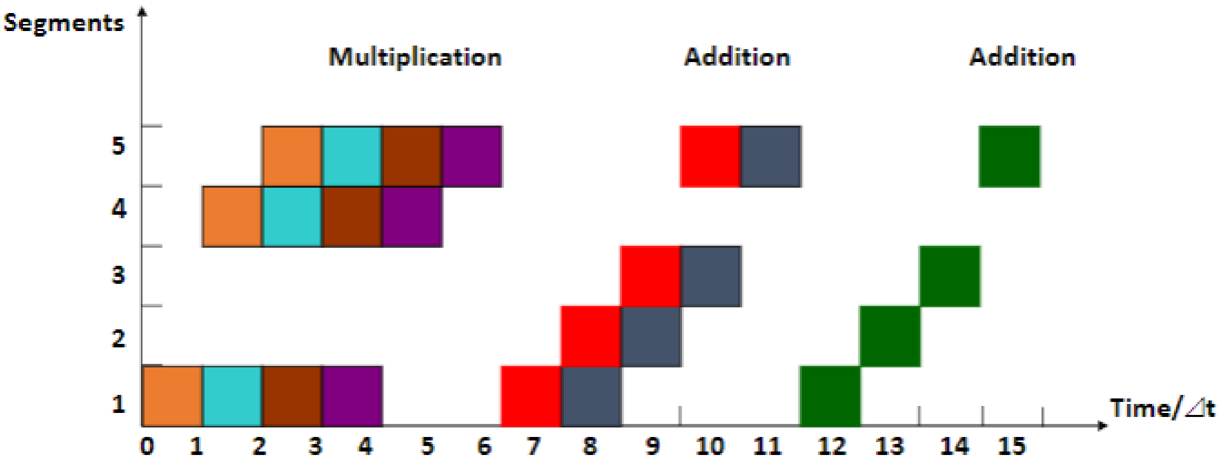
Example



Pipeline Performance

- Vector A(a1, a2,a3,a4)
- Vector B(b1,b2,b3,b4)
- Compute vector dot product (A·B) in the static dual-function pipelining.

- 1→2→3→5 Addition pipelining
- 1→4→5 Multiplication pipelining
- The time of each segment in the pipelining is ⊿t。

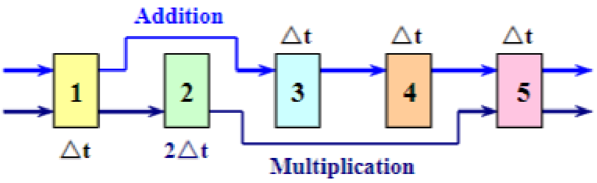


TP=7/(15⊿t)=0.47/⊿t

Sp=(4×3⊿t+3×4⊿t)/(15⊿t)=1.6

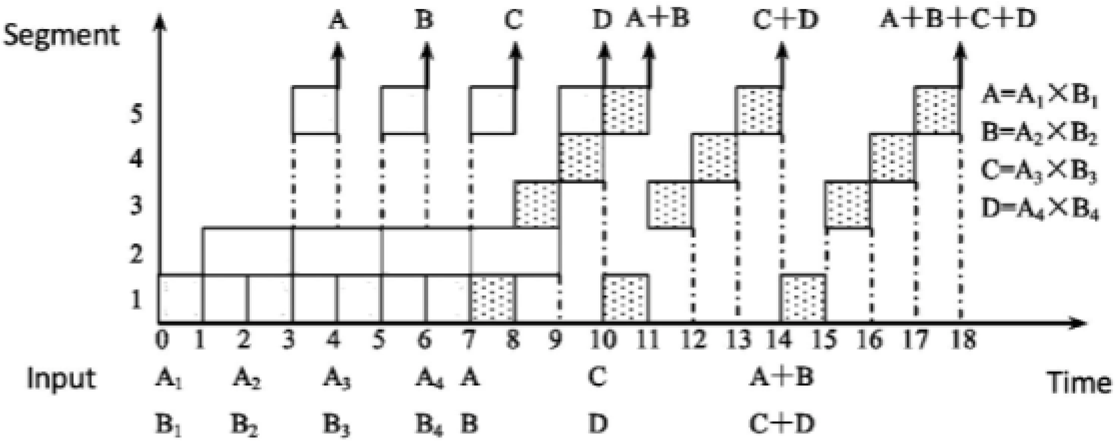
η=(3×4⊿t+4×3⊿t)/(5×15⊿t)=32%

- Vector A(a1, a2,a3,a4)
- Vector B(b1,b2,b3,b4)
- Compute vector dot product (A·B) in the dynamic dual-function pipelineing



1→3→4→5 Addition pipelining

1→2→5 Multiplication pipelining



$$TP=\frac{7}{18\Delta t} \qquad S=\frac{28\Delta t}{18\Delta t}\approx 1.56 \qquad E=\frac{4\times 4+3\times 4}{5\times 18}\approx 0.31$$

Ideal Performance for Pipelining

- If the stages are perfectly balanced, the time per instruction on the pipelined processor equal to: $\frac{Time\ per\ instruction\ on\ unpipelined\ machine}{Number\ of\ pipelined\ stages}$
- So ideal speedup equal to Number of pipeline stages

RISC-V Pipelining

- Five stages, one step per stage

1. IF: 取指，取出 I-Mem 中 PC 地址处的指令
2. ID: 译码，将指令解码为控制信号，并读取寄存器值
3. EX: 执行，执行 ALU 操作
4. MEM: 访存，访问 D-Mem 进行写入或读取
5. WB: 写回，将结果写回寄存器文件

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - 指令长度固定为 32 位，易于在一个周期内进行取指或译码
  - 指令格式少且规整，易于在一个周期内译码、读取寄存器
  - 使用 load/store 寻址结构，一个周期计算地址、一个周期访存
  - 访存操作都是对齐的，可以在一个周期内进行

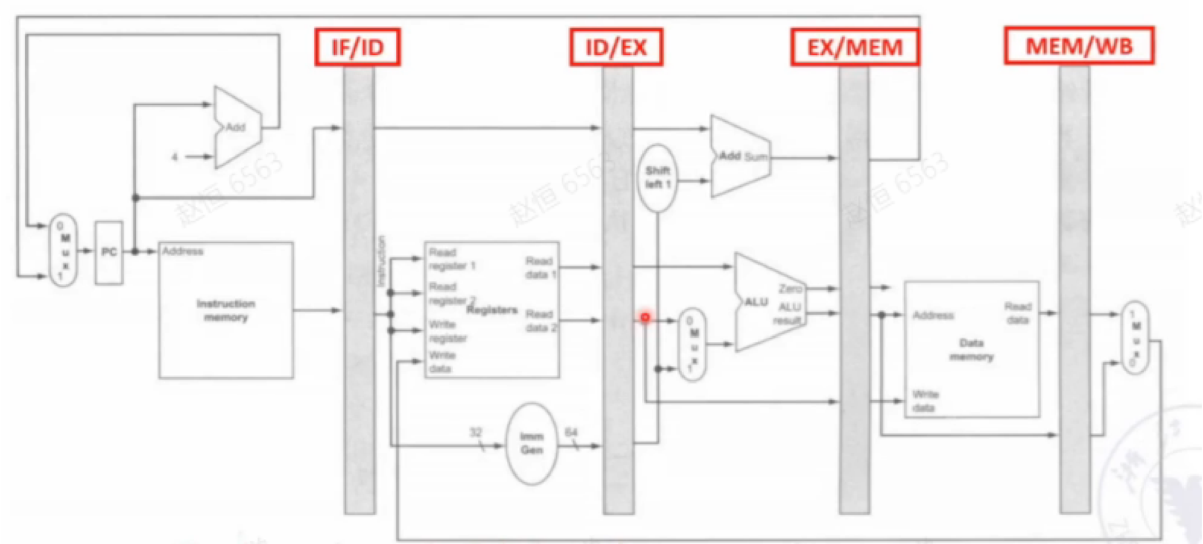
An Implementation of Pipelining

- 添加阶段寄存器（pipeline register）来分隔每个阶段：IF/ID、ID/EX、EX/MEM、MEM/WB
  - 这四个阶段寄存器和 PC 寄存器一起将流水线分为了五个部分
  - 可以**看作**只有这五个时序电路，其它内部操作都是组合逻辑，在内部运行
  - 五个寄存器在上升沿进行更新，阶段寄存器进行流转，记录当前指令需要的信息
- 数据通路中有两个回路
  - MEM 阶段计算分支结果，输出给 PC。可能会引起控制冒险
  - WB 阶段写回寄存器，可能会引起数据冒险
- ID 和 WB 阶段同时使用寄存器组文件，但不会产生结构冒险，因为 ID 阶段只读取寄存器，WB 阶段只写入寄存器，相当于分为了两个部分

How Pipelining Improves Performance

- Decreasing the execution time of an individual instruction ✗
- Increasing instruction throughput ✓

The Pipelined Version of the Datapath

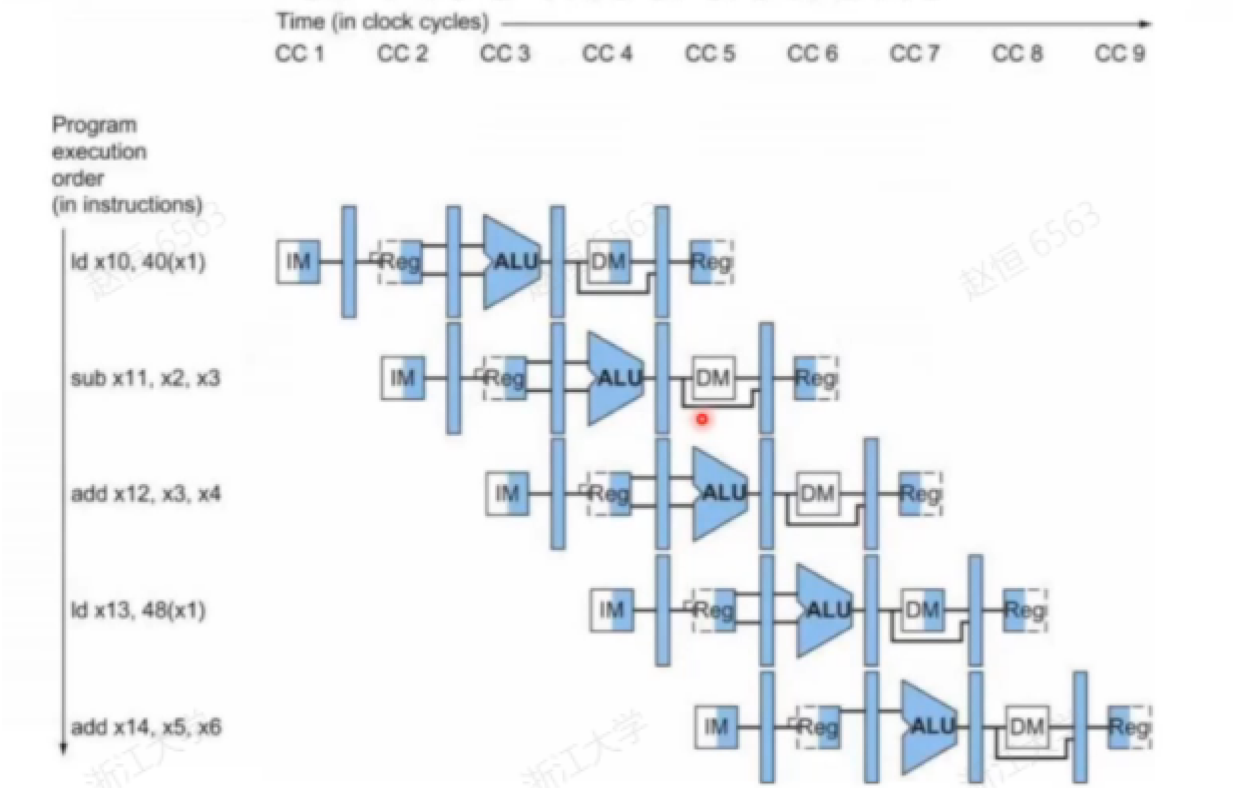


Each register hold data from previous stage

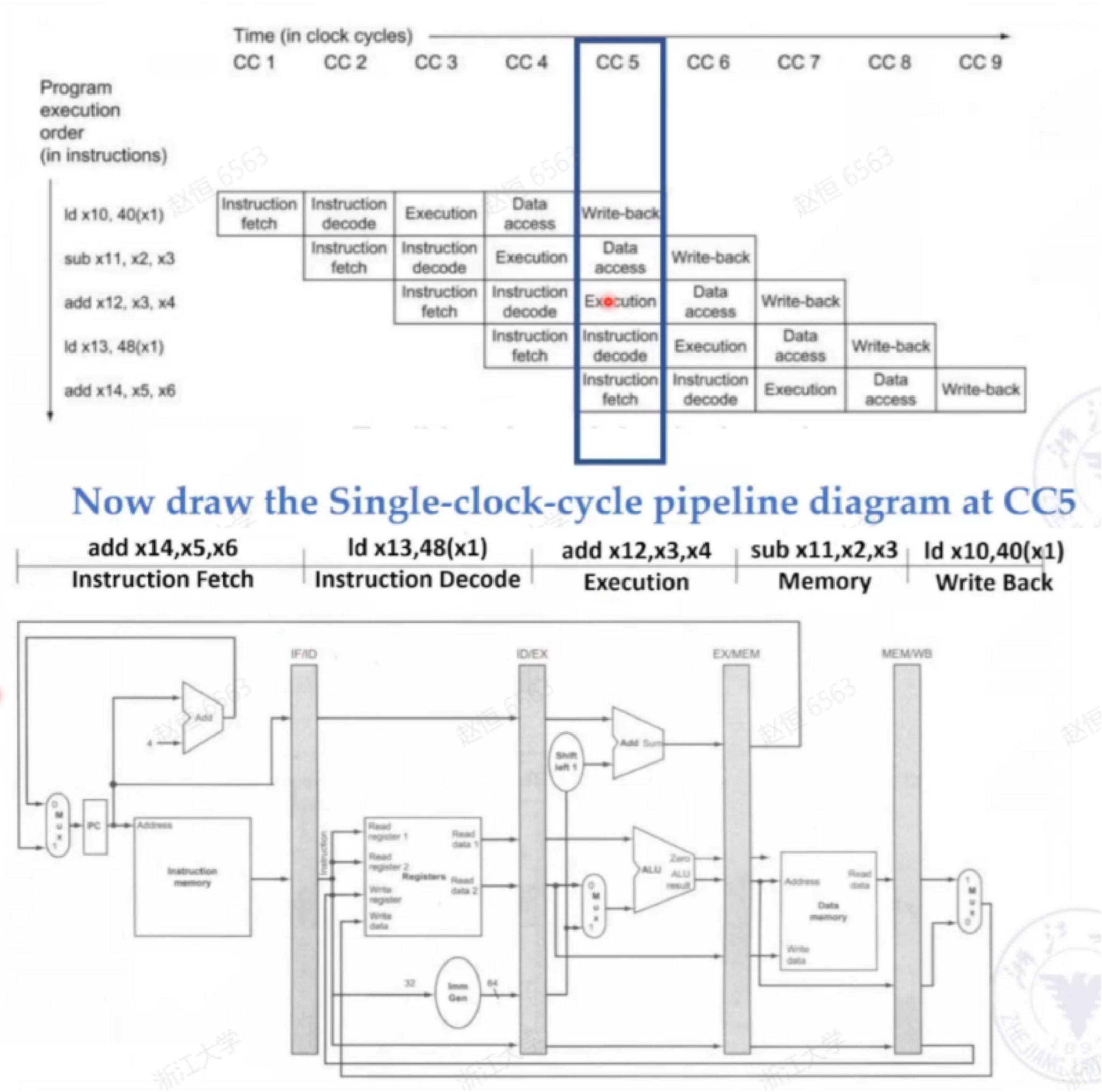
- 几种呈现方式：

> 了解即可，考试不要求绘画

- Multiple-Clock-Cycle Pipeline Diagram of five Instructions



- Traditional Multiple-Clock-Cycle Pipeline Diagram



Time (in clock cycles)

Program execution order (in instructions)

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
ld x10, 40(x1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub x11, x2, x3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add x12, x3, x4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
ld x13, 48(x1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add x14, x5, x6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Now draw the Single-clock-cycle pipeline diagram at CC5

add x14,x5,x6	ld x13,48(x1)	add x12,x3,x4	sub x11,x2,x3	ld x10,40(x1)
Instruction Fetch	Instruction Decode	Execution	Memory	Write Back

Pipeline Hazards

Structural Hazard | 结构冒险

- A required resource is busy
- Solution:
  1. Instructions take it in turns use resource, some instruction have to stall
  2. Add more hardware to machine

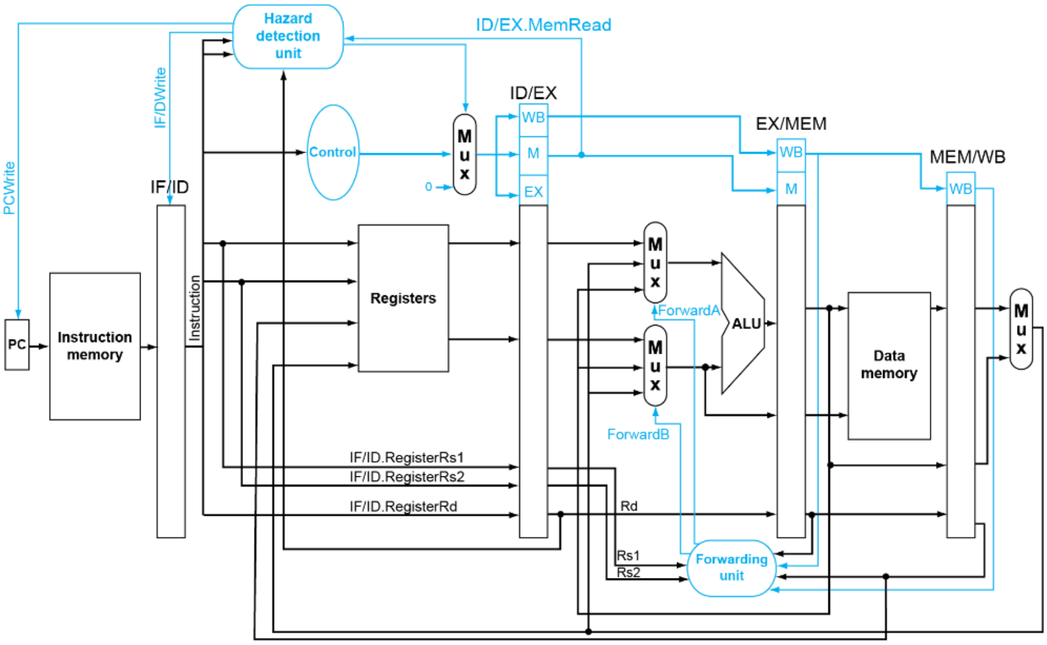
> Can always solve by adding more hardware

Data Hazards | 数据冒险

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write
- Solution:
    - Forwarding | 前递: Adding extra hardware to retrieve the missing item early from the internall resources
        - Forwarding Conditions:

Mux Control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register
FA = 10	EX/MEM	~ comes from the prior ALU result
FA = 01	MEM/WB	~ comes from data memory or an earlier ALU result
FB = 00	ID/EX	The second ALU operand ~
FB = 10	EX/MEM	~ ~
FB = 01	MEM/WB	~ ~

    - Stall | 暂停:
        - 对于 Load-Use Data Hazard
            - 在 ID 阶段就进行探测 load-use 冒险
                - ID/EX.MemRead == 1 (ID/EX.MemWrite == 0)
                - ID/EX.Rd == IF/ID.Rs1 或 IF/ID.Rs2
            - 暂停流水线:
                - 强制 ID/EX 阶段寄存器中的控制信号变为 0 (相当于插入一条 nop)
                - 阻止 PC 寄存器和 IF/ID 阶段寄存器更新
            - 在暂停一个周期后就可以按照 MEM hazard 进行前递解决
    - DataPath with Hazard Detection



Control Hazards | 控制冒险

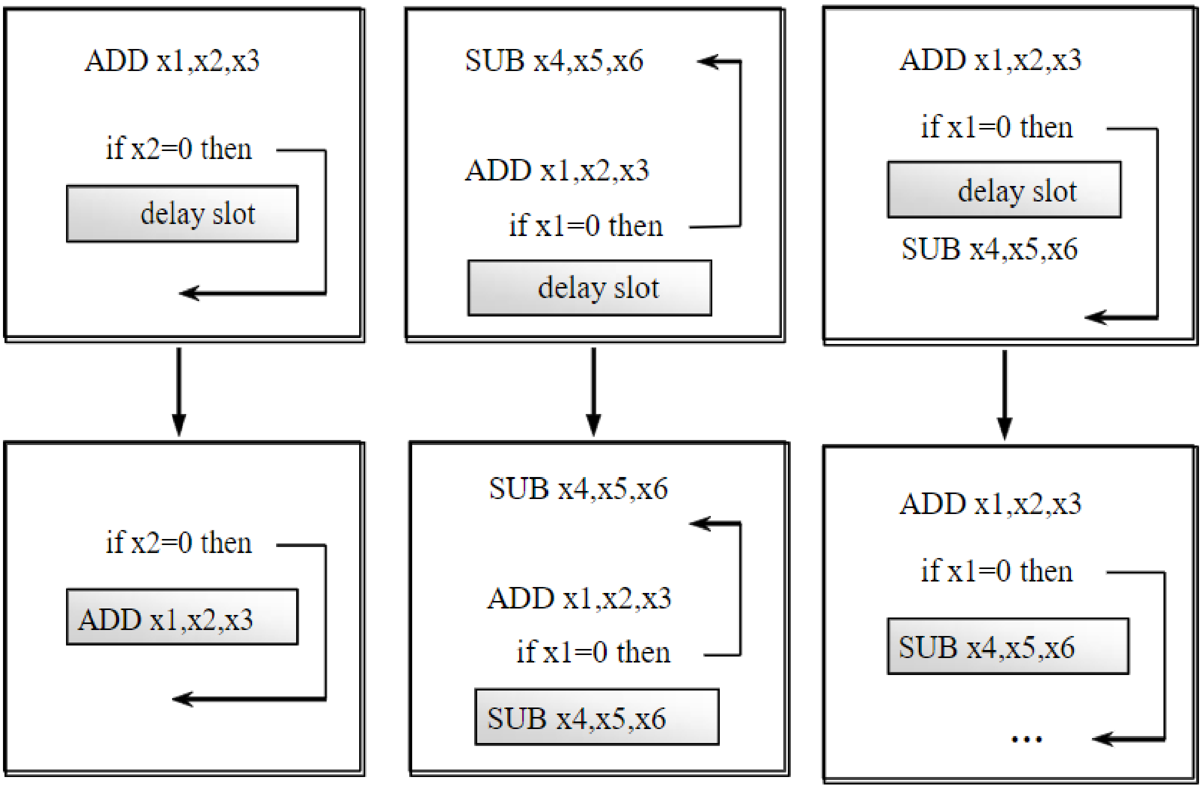
- Flow of execution depends on previous instruction
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipelining can’t always fetch correct instruction
    - Still working on ID stage of branch
- In RISC-V pipelining
  - Need to compare registers and compute target early in the pipelining
  - Add hardware to do it in ID stage
- How to Reduce Branch Delay

> - Key processes in branch instructions
>   - Compute the branch target address
>   - Judge if the branch success

  - Move hardware to determine outcome to ID stage
    - Target adddress adder
    - Register comparator

Code Scheduling

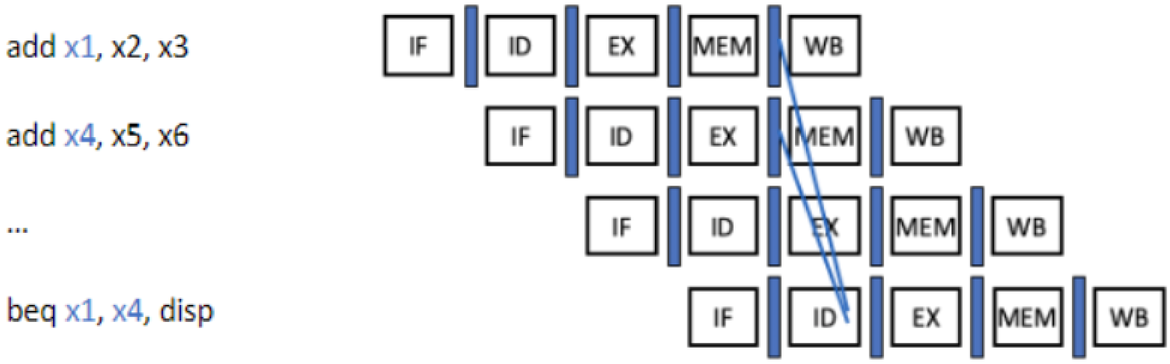
> 分支延迟槽，来避免一些冲突(MIPs常使用)



Scheduling from former   Scheduling from object   Scheduling from failure

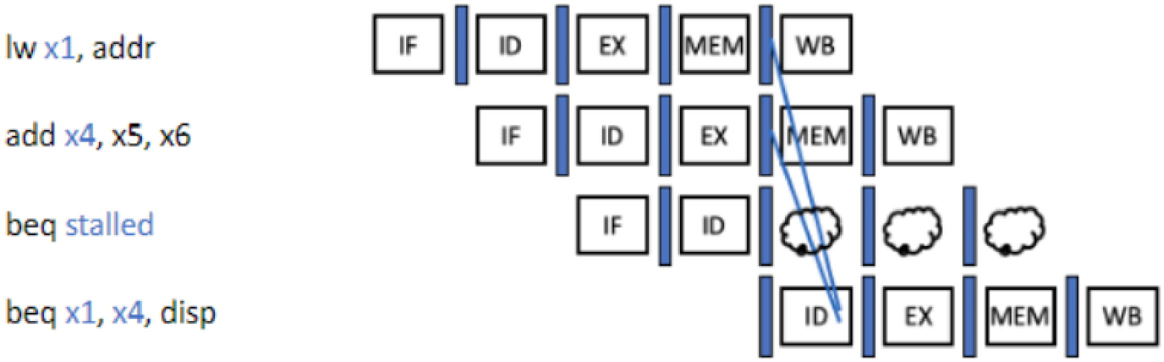
Data Hazards for Branches

- If a comparsion register is destination of 2nd or 3rd preceding ALU instruction



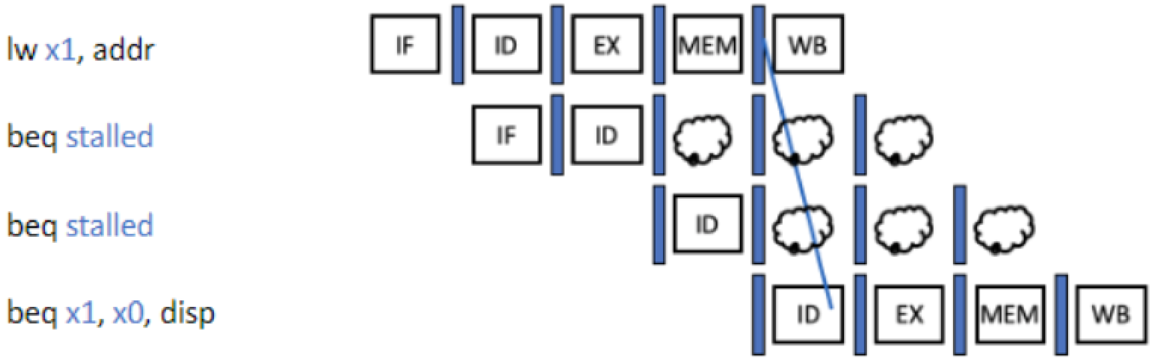
  - Can resolve using forwarding

- If a comparsion register is a destination of preceding ALU instruction or 2nd preceding load instruction



  - Need 1 stall cycle

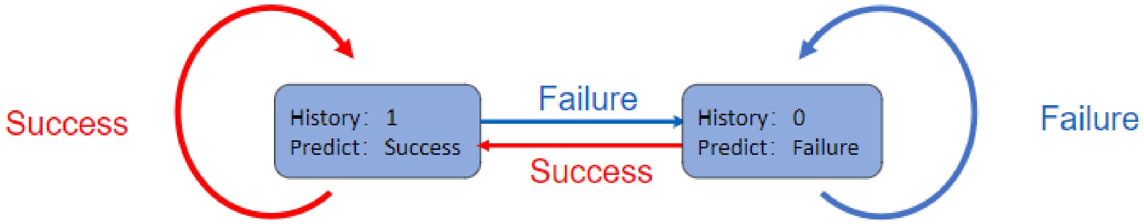
- If a comparsion register is a destination of immediately preceding load instruction



  - Need 2 stall cycles

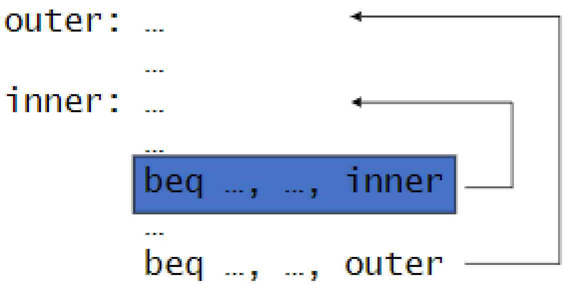
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table | 分支历史表)
  - Indexed by recent branch instruction addresses | 表头是最近的分支指令的地址
  - Stores outcome (taken/not taken) | 表的内容是是否跳转
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction



1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice! | 内循环结束和再次进入的时候有两个误预判



  - Mispredict as taken on last iteration of inner loop

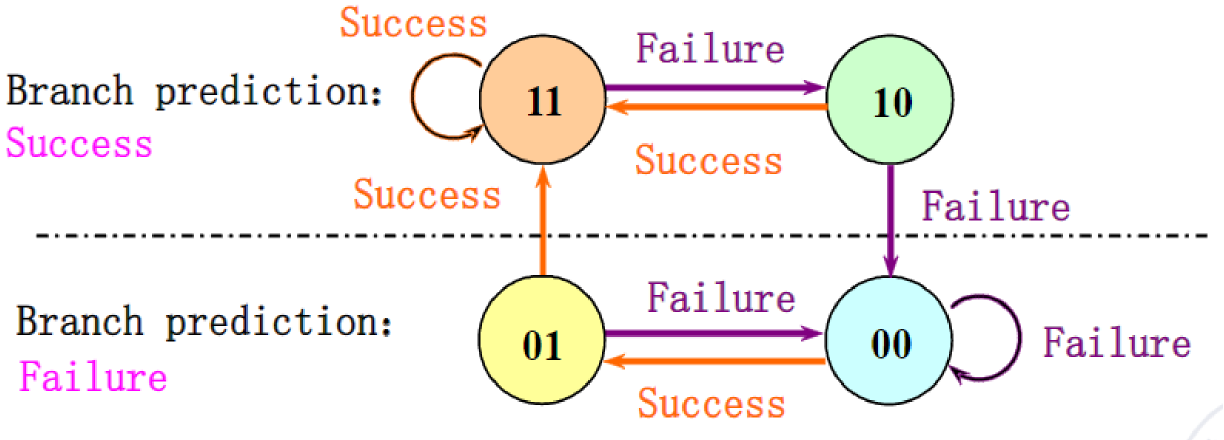
- Then mispredict as not taken on first iteration of inner loop next time around

  > 为解决这个问题，引入两位的预测器

2-Bit Predictor

- Only change prediction on two successive mispredictions

> 有限状态机



Instruction-Level Parallelism(ILP)

> 多发射,在一个时钟周期发射多条指令，同步执行

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage → shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages → multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle(IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in pratice

Multiple Issus

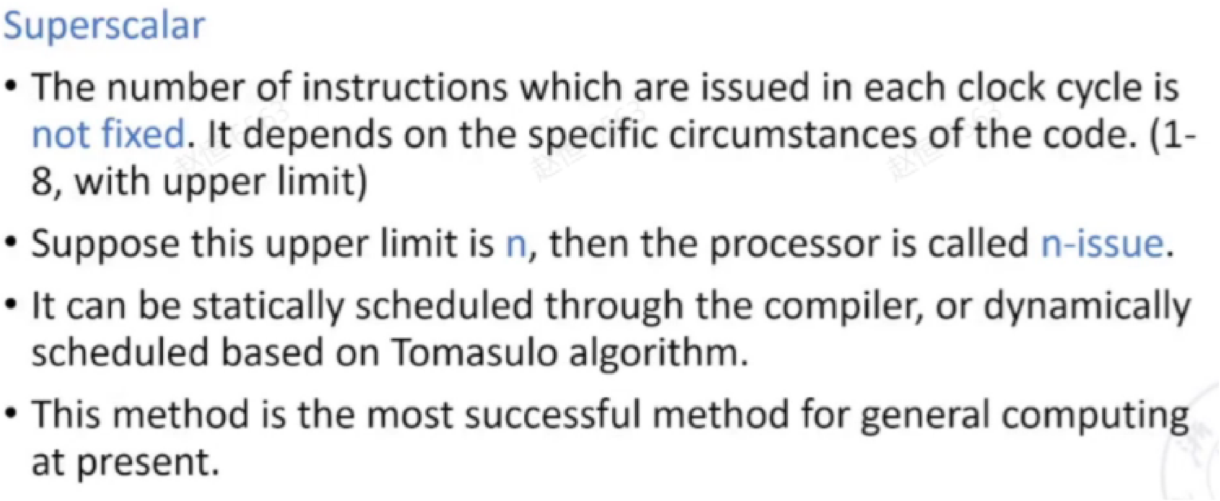
- Static multiple issue | 静态多发射

  > 由软件（编译器）决定哪些指令发射出去

  - Compiler groups instructions to be issued together
  - Packages them into “issue slots (发射槽) ”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by recordering instructions
  - CPU resolves hazards suing advanced techniques at run time

Two types of multiple-issue processor

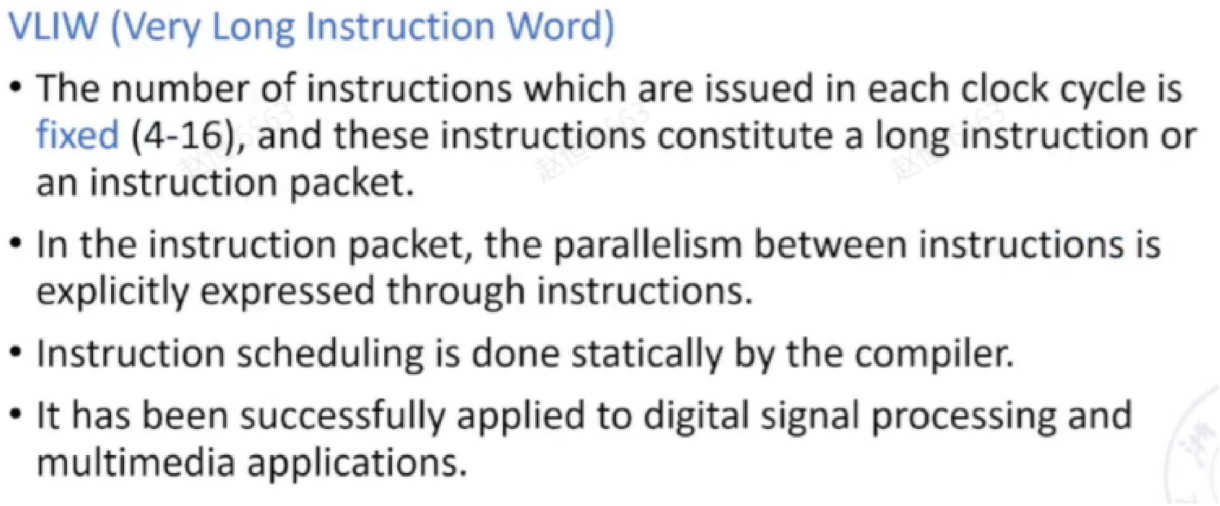
- 和多发射基本相同，但每次发射的条数是不固定的



Superscalar

- The number of instructions which are issued in each clock cycle is not fixed. It depends on the specific circumstances of the code. (1-8, with upper limit)
- Suppose this upper limit is n, then the processor is called n-issue.
- It can be statically scheduled through the compiler, or dynamically scheduled based on Tomasulo algorithm.
- This method is the most successful method for general computing at present.

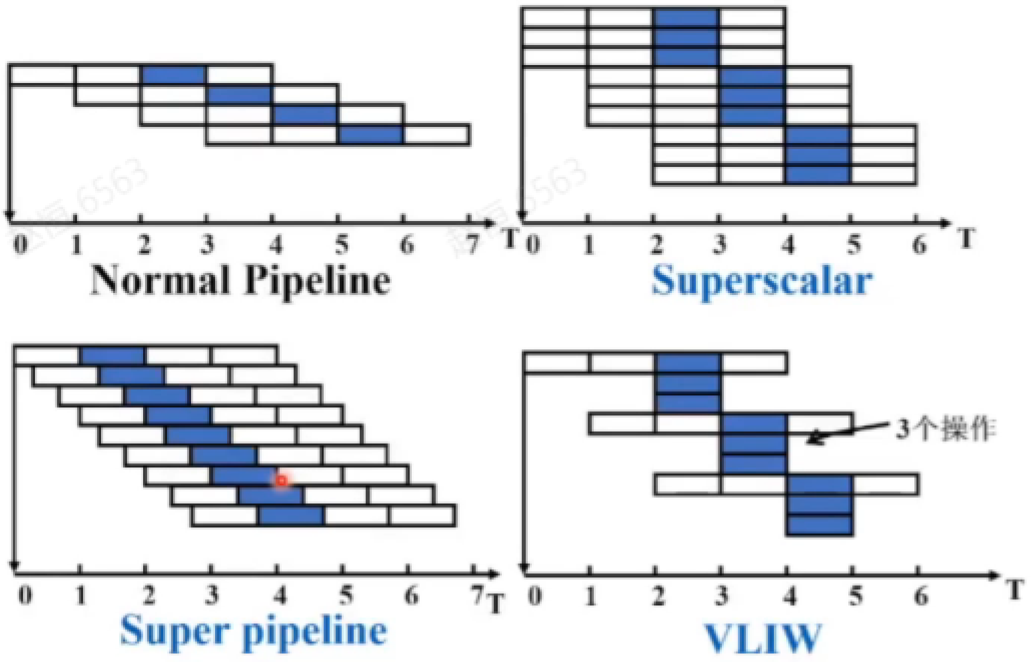
- 超长指令字 | 静多发射一定发射条数，由编译器管理



VLIW (Very Long Instruction Word)

- The number of instructions which are issued in each clock cycle is fixed (4-16), and these instructions constitute a long instruction or an instruction packet.
- In the instruction packet, the parallelism between instructions is explicitly expressed through instructions.
- Instruction scheduling is done statically by the compiler.
- It has been successfully applied to digital signal processing and multimedia applications.

Comparison

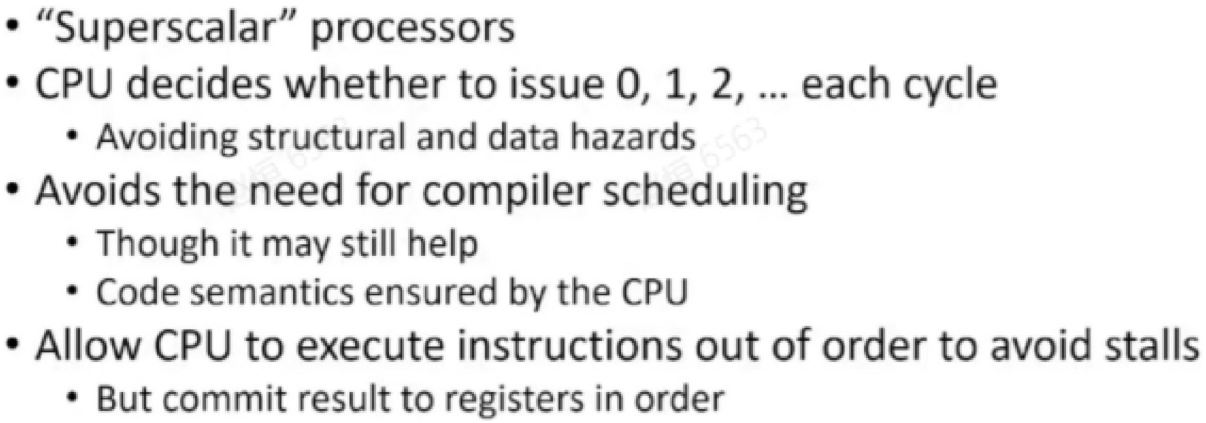


Scheduling

Static Multiple Issue

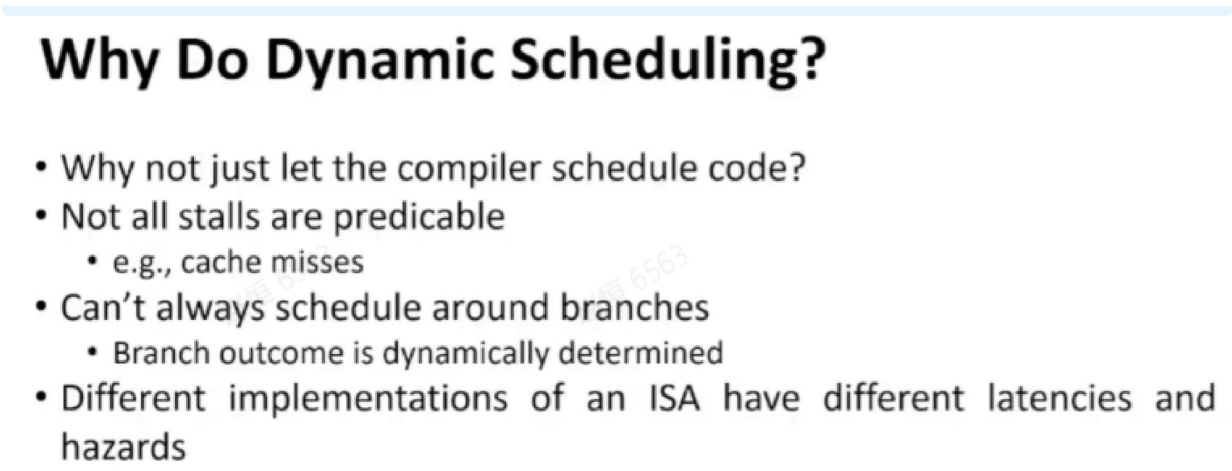
- Compiler must remove some/all hazards
  - Recorder instructions inti issue packeys
  - No dependencies with a packet
  - Possibly some dependecies between packets
    - Varies between ISAs; compiler must konw!
  - Pad with nop if necessary

Dynamic Multiple Issue



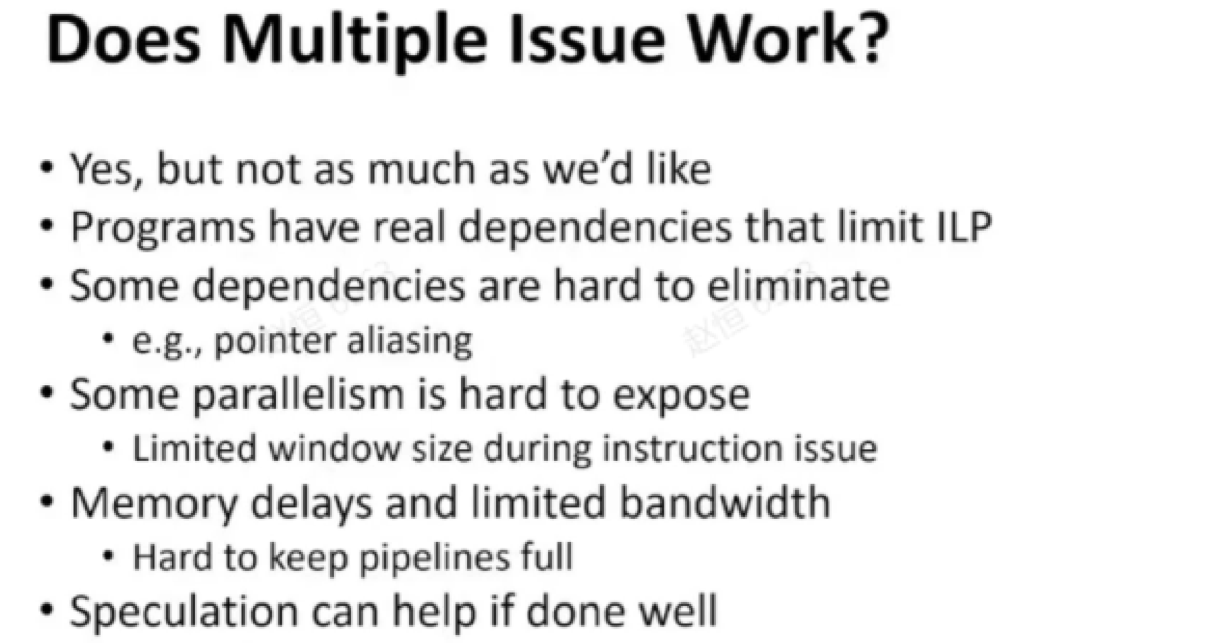
- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
- Allow CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order

一些性质



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards



Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well