处理器体系结构

Processor Architecture

本章内容

Topic

- 指令集体系结构

  Instruction Set Architecture

- 顺序执行CPU的实现

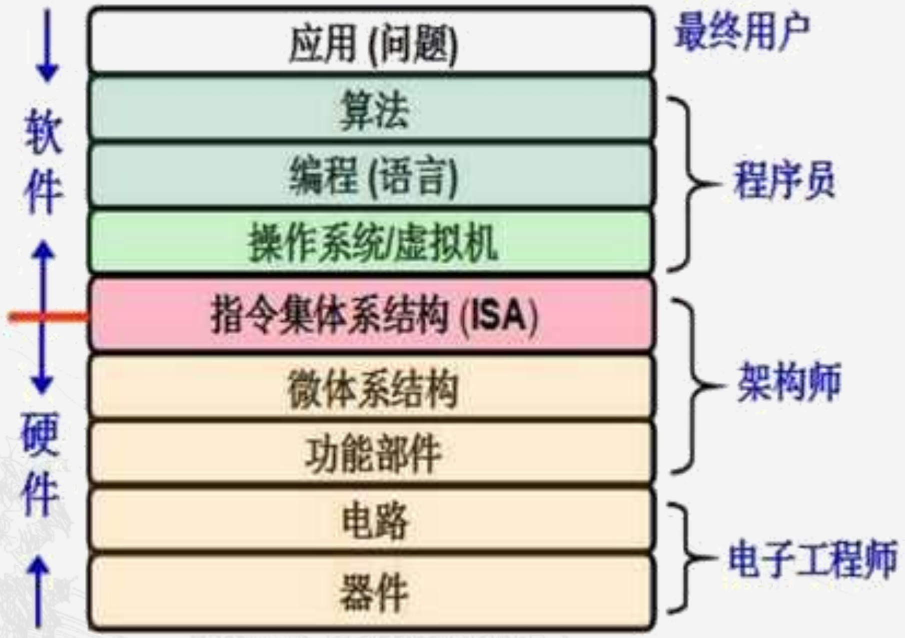
  Sequential CPU Implementation

- 流水线原理

  Principle of Pipeline

指令集体系结构

Instruction Set Architecture



- 什么是指令集体系结构?
  What is it ?
  - 汇编语言的抽象
    Assemble Language Abstraction
  - 机器语言的抽象
    Machine Language Abstraction
- 一种真实计算机的抽象，隐藏了实现细节
  An abstraction of the real computer, hide the details of implementation
  - 计算机指令的语法
    The syntax of computer instructions
  - 指令的语义
    The semantics of instructions
  - 执行模型
    The execution model
  - 程序员可见的计算机状态
    Programmer-visible computer status
- 是一种软件和硬件之间的接口规范
  An interface specification between software and hardware

指令集体系结构
Instruction Set Architecture

不同的家族，不同的指令集体系结构
“Different families” have different ISAs

- 复杂指令集计算机 （CISC）
  Complex instruction set computer (CISC)
  - x86家族：IA32(x86-32)，x86-64
    x86 families: IA32 (x86-32)，x86-64
  - System/360　PDP-11　VAX　Data General Nova

  - 嵌入式处理器：Motorola 6800, Zilog Z80, 8051-family
    Embedded processors: Motorola 6800, Zilog Z80, 8051-family

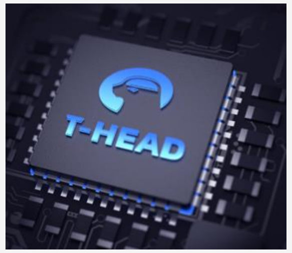
指令集体系结构

Instruction Set Architecture

不同的家族，不同的指令集体系结构
“Different families” have different ISAs

- 精简指令集计算机 （RISC）
  Reduced instruction set computer (RISC)
  - IBM/Freescale Power　　ARM　　MIPS、LoongISA

  - SPARC　　RISC-V



复杂指令集 （以IA32为例）
CISC Instruction Sets （IA32 Example）

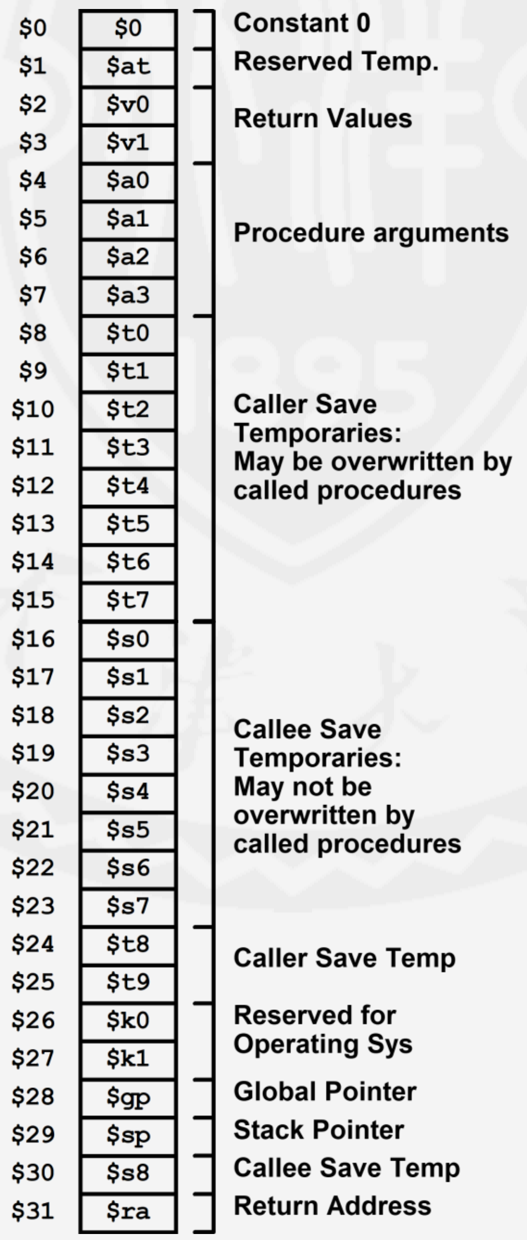
- 面向栈的指令集
  Stack-oriented instruction set
  - 使用栈传递参数，保存程序计数器
    Use stack to pass arguments, save program counter
  - 显式的入栈和出栈指令
    Explicit push and pop instructions
- 算术运算指令可以直接访问内存
  Arithmetic instructions can access memory
  - `addq %rax, 12(%rbx,%rcx,8)`
    - 包含了存储器的读和写
      requires memory read and write
    - 包含了复杂的地址计算
      Complex address calculation
- 条件码
  Condition codes
  - 可以通过算术逻辑运算的指令的副作用设置
    Set as side effect of arithmetic and logical instructions
- 设计哲学
  Philosophy
  - 使用指令实现典型的任务
    Add instructions to perform “typical” programming tasks

指令集体系结构

Instruction Set Architecture

精简指令集 （以MIPS为例）
RISC Instruction Sets （MIPS Example）

- 更少的，更简单的指令
  Fewer, simpler instructions
  - 需要花费更多的指令完成给定的任务
    Might take more to get given task done
  - 可以在更小更快的硬件上执行
    Can execute them with small and fast hardware
- 面向寄存器的指令集
  Register-oriented instruction set
  - 更多的寄存器 （典型值：32）
    Many more (typically 32) registers
  - 用于传递参数，返回地址，临时数据
    Use for arguments, return pointer, temporaries
- 只有加载和存储指令可以访问内存
  Only load and store instructions can access memory
  - `lw  $t1, 0($s0)`
  - `sw  $s0, 0($sp)`
- 没有条件码
  No Condition codes
  - 测试指令将返回结果0/1写入寄存器
    Test instructions return 0/1 in register



\$0	\$0	Constant 0
\$1	\$at	Reserved Temp.
\$2	\$v0	Return Values
\$3	\$v1	
\$4	\$a0	Procedure arguments
\$5	\$a1	
\$6	\$a2	
\$7	\$a3	
\$8	\$t0	Caller Save Temporaries: May be overwritten by called procedures
\$9	\$t1	
\$10	\$t2	
\$11	\$t3	
\$12	\$t4	
\$13	\$t5	
\$14	\$t6	
\$15	\$t7	
\$16	\$s0	Callee Save Temporaries: May not be overwritten by called procedures
\$17	\$s1	
\$18	\$s2	
\$19	\$s3	
\$20	\$s4	
\$21	\$s5	
\$22	\$s6	
\$23	\$s7	
\$24	\$t8	Caller Save Temp
\$25	\$t9	
\$26	\$k0	Reserved for Operating Sys
\$27	\$k1	
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$s8	Callee Save Temp
\$31	\$ra	Return Address

MIPS registers

指令集体系结构

Instruction Set Architecture

CISC vs. RISC

- 出发点
  Original Debate
  - CISC：更简单的编译器，更少的字节码
    CISC: easy for compiler, fewer code bytes
  - RISC：更强大的编译优化，可以在更加简单的硬件上面快速运行
    RISC: better for optimizing compilers, can make run fast with simple chip design

- 当前状态
  Current Status
  - 桌面平台ISA的选择从来不是一个技术问题
    For desktop processors, choice of ISA not a technical issue
    - 有足够的硬件资源，可以使指令运行得很快
      With enough hardware, can make anything run fast
    - 代码的兼容性是更重要的问题
      Code compatibility more important
  - X86-64借鉴了很多RISC的特征
    x86-64 adopted many RISC features
    - 更多的寄存器，用来传递参数
      Programmer-visible computer status

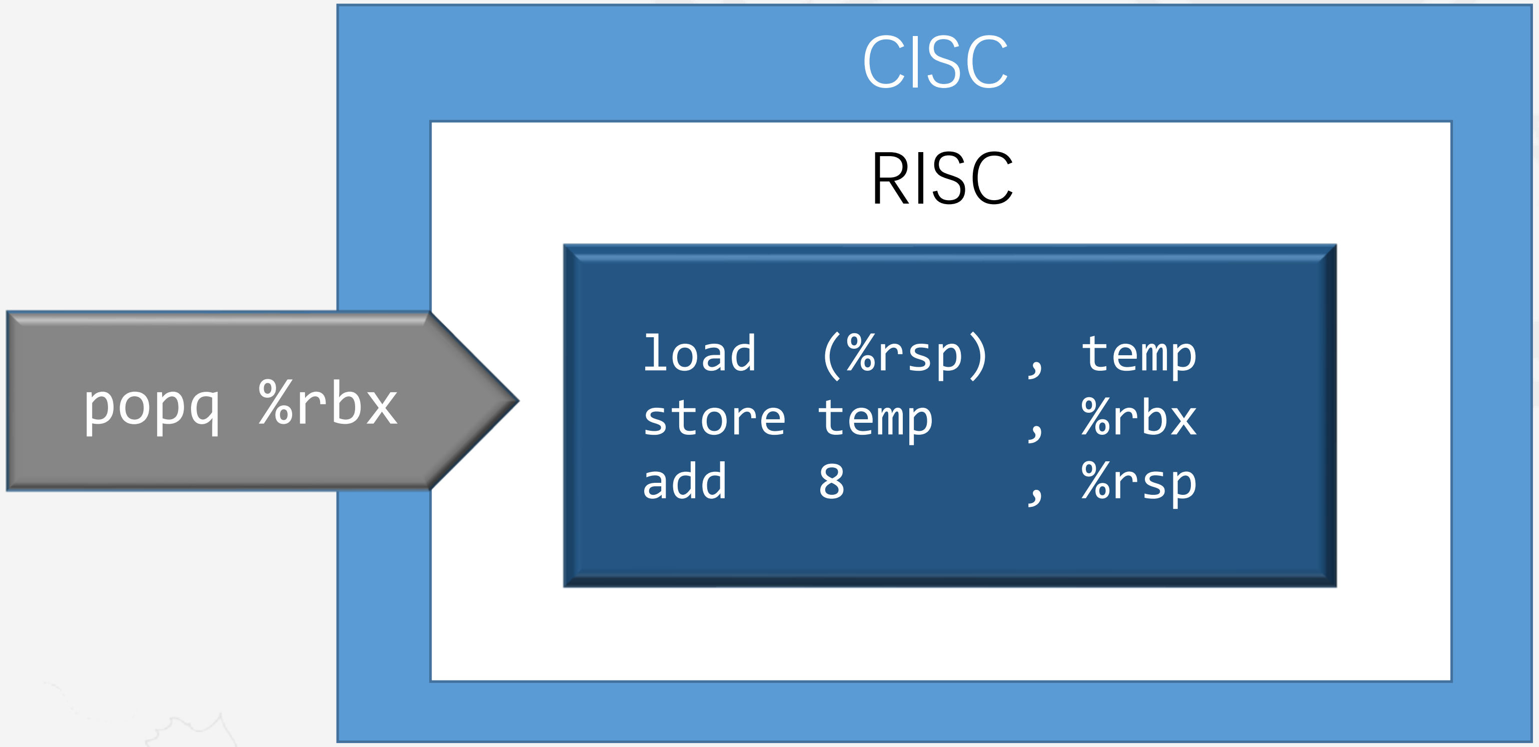
- 对于嵌入式处理器，RISC更有意义
  For embedded processors, RISC makes sense
  - 硬件资源更少、价格更低、功耗更低
    Smaller, cheaper, less power
  - 绝大多数的手机使用的都是ARM处理器
    Most cell phones use ARM processor

指令集体系结构

Instruction Set Architecture

X86处理器的实现细节
Implementation of x86 processor in detail

- 微指令和微程序
  Microcode & Micorprogram
  - x86 是 CISC
    x86 is CISC
  - 但仅有一个CICS的壳
    But only the shell is CISC
  - 内部核心是RISC的
    The core is RISC



```
load   (%rsp) , temp
store  temp   , %rbx
add    8      , %rsp
```

本章内容

Topic

- ☐ 指令集体系结构

  Instruction Set Architecture

- ☐ 顺序执行CPU的实现

  Sequential CPU Implementation

- ☐ 流水线原理

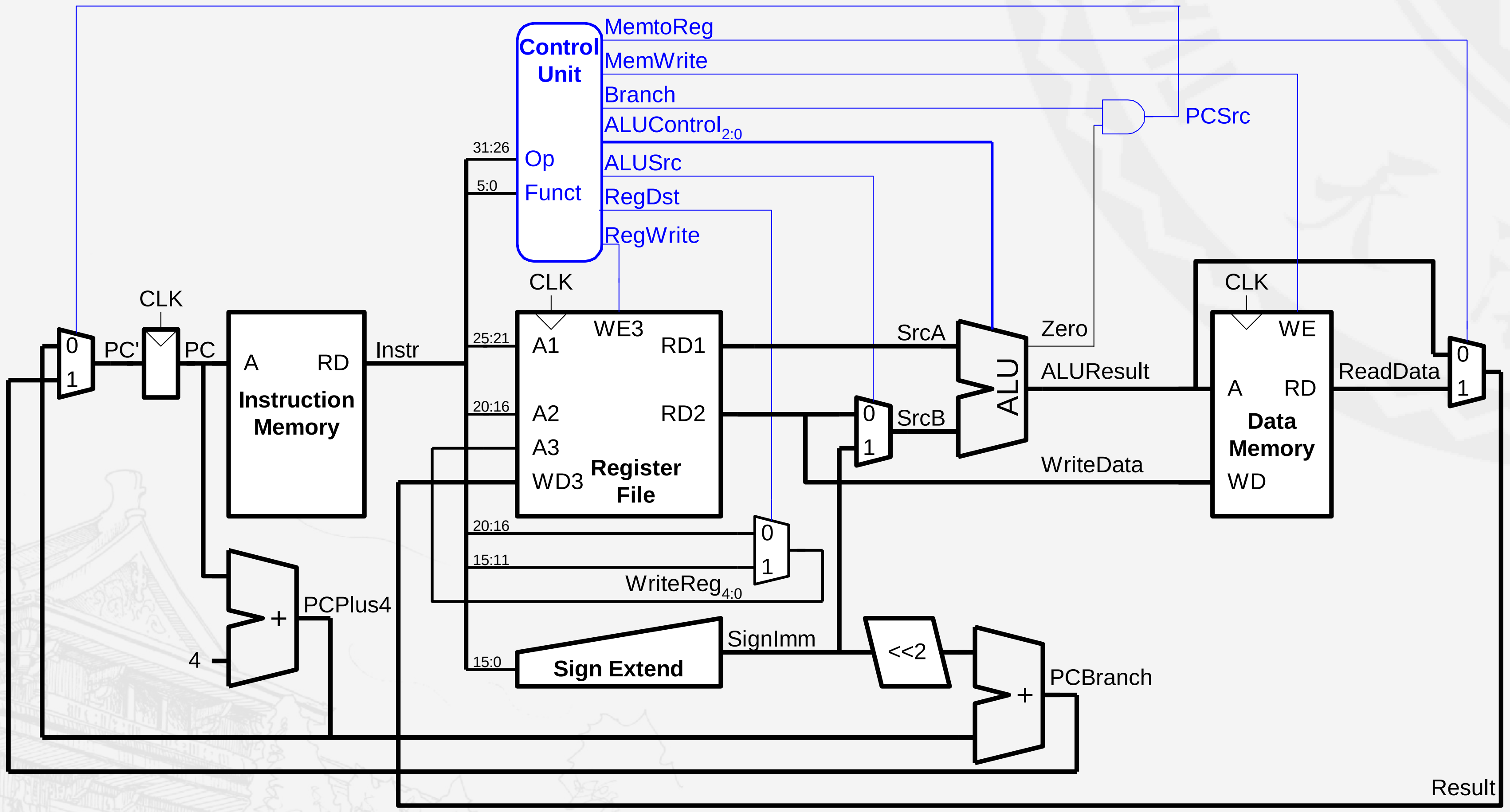
  Principle of Pipeline

顺序执行CPU的实现

Sequential CPU Implementation

大类基础课"数字逻辑与数字系统"的综合实验

一个顺序执行的MIPS处理器的设计图
A Sequential MIPS Design Diagram



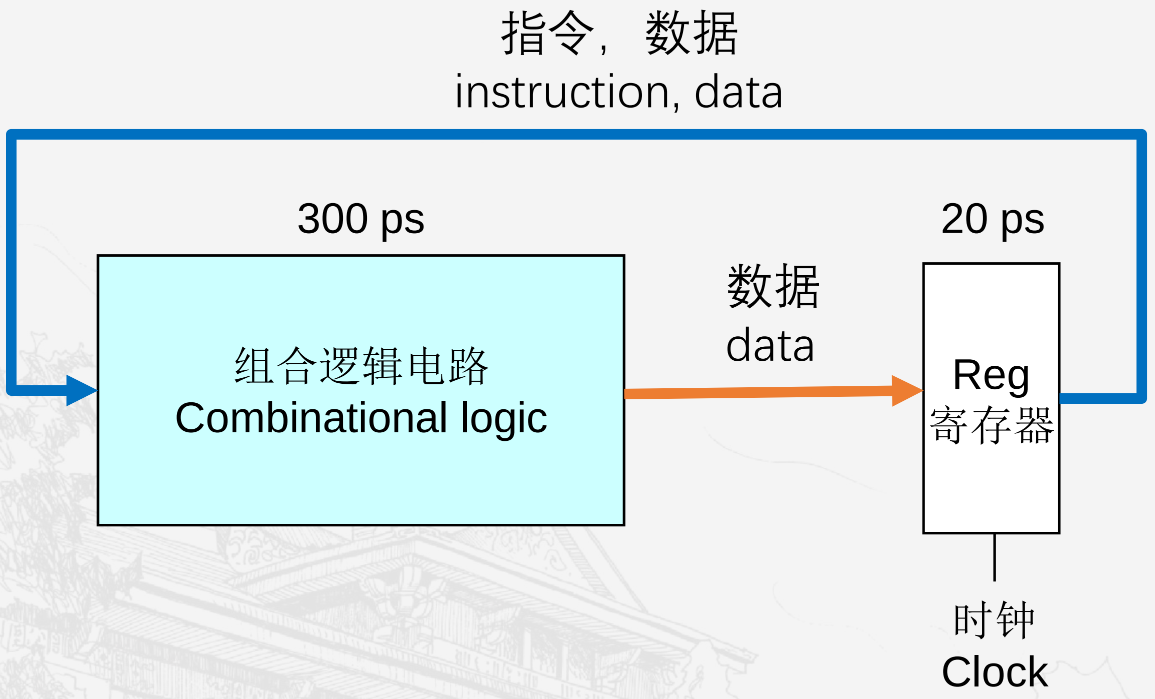
顺序执行CPU的实现

Sequential CPU Implementation

ps：皮秒

1ps = 10^{-12}s

一个顺序执行的处理器原型
Prototype of a Sequential CPU



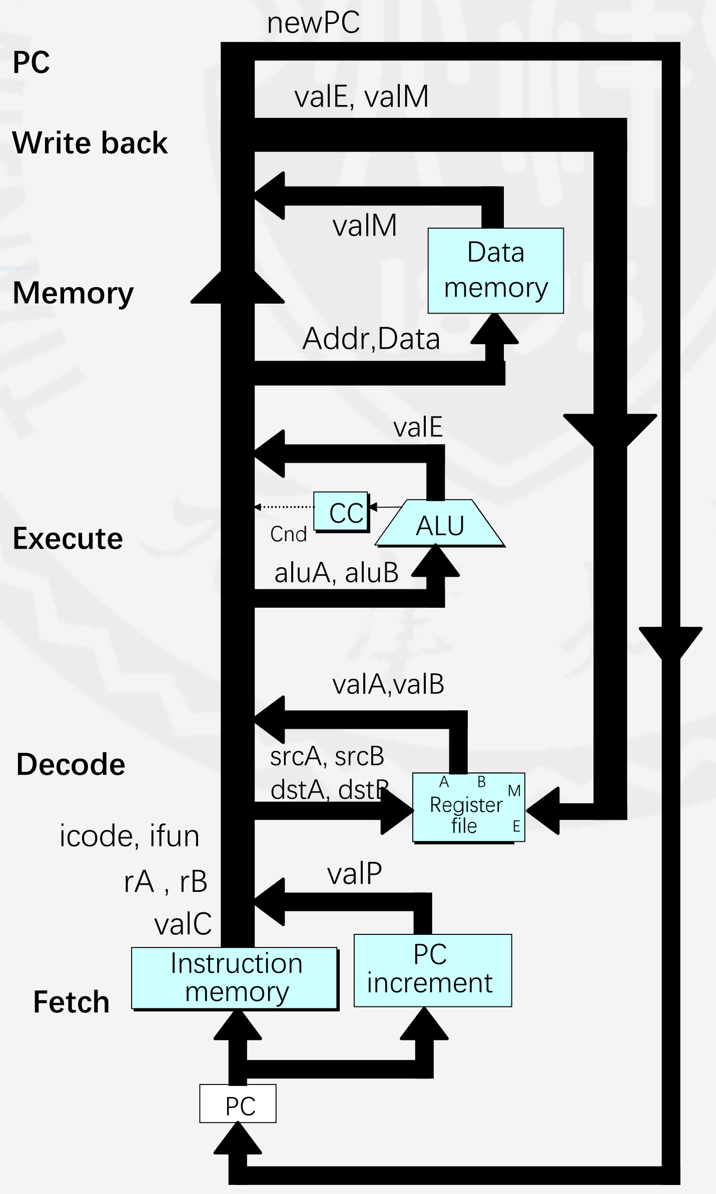
- 系统
  System
  - （组合逻辑电路）计算需要300ps
    Computation requires total of 300 picoseconds
  - 额外的20ps用于将结果存储至寄存器（内存）
    Additional 20 picoseconds to save result in register (memory)
  - 时钟周期不能小于320ps
    Can must have clock cycle of at least 320 ps

顺序执行CPU的实现

Sequential CPU Implementation

指令执行的阶段
Instruction Execution Stages

1. **取指**：从指令存储器读取指令
   Fetch: Read instruction from instruction memory
2. **译码**：读取程序所需的寄存器
   Decode: Read program registers
3. **执行**：计算值和地址
   Execute: Compute value or address
4. **仿存**：从内存读数据或向内存写数据
   Memory: Read or write data
5. **写回**：写程序相关的寄存器
   Write Back: Write program registers
6. **更新PC**：更新程序计数器
   PC: Update program counter



顺序执行CPU的实现

Sequential CPU Implementation

举例：addq 指令的执行 （寄存器之间）
Executing addq Operation Example (between registers)

addq rA， rB

1. **取指**：读3个字节
   Fetch: Read 3 bytes
2. **译码**：读取操作数的寄存器rA和rB
   Decode: Read program registers rA and rB
3. **执行**：执行加法运算，设置条件码
   Execute: Perform add operation
   Set condition codes
4. **仿存**：什么都不做
   Memory: Do nothing
5. **写回**：更新寄存器 rB
   Write Back: Update register rB
6. **更新PC**：程序计数器加 3
   PC: Increment PC by 3

顺序执行CPU的实现

Sequential CPU Implementation

举例：movq指令的执行（从内存到寄存器）
Executing movq Operation Example (from mem to reg)

movq D(rB), rA

1. **取指**：读4个字节
   Fetch: Read 4 bytes
2. **译码**：读取操作数的寄存器rB
   Decode: Read program registers rB
3. **执行**：计算有效地址
   Execute: Compute effective address
4. **仿存**：读取内存
   Memory: Read from memory
5. **写回**：更新寄存器 rA
   Write Back: Update register rA
6. **更新PC**：程序计数器加 4
   PC: Increment PC by 4

顺序执行CPU的实现

Sequential CPU Implementation

顺序执行处理器的问题
Problem of SEQ

- 性能差
  Too slow
  - 在一个时钟周期内需要做太多的事情
    Too many tasks needed to finish in one clock cycle
  - 信号传播全部的指令阶段需要较长的时间
    Signals need long time to propagate through all of the stages
  - 时钟周期需要足够大（才能够保证逻辑正确）
    The clock must run slowly enough
- 没有充分的利用硬件中的各个处理单元
  Does not make good use of hardware units
  - 在整个周期中，每个处理单元只有部分时间处于活跃状态
    Every unit is active for part of the total clock cycle

本章内容

Topic

- 指令集体系结构

  Instruction Set Architecture

- 顺序执行CPU的实现

  Sequential CPU Implementation

- 流水线原理

  Principle of Pipeline

流水线原理

Principle of Pipeline

现实世界的流水线：洗车
Real-World Pipelines: Car Washes

顺序
Sequential

并行
Parallel

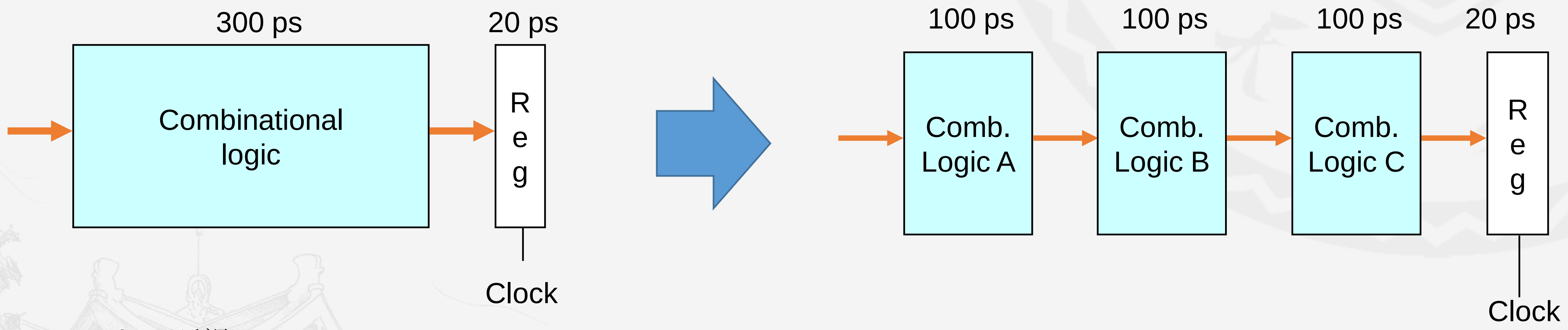
流水线
Pipelined

- 思想
  Idea
  - 将过程划分成多个独立阶段
    Divide process into independent stages
  - 对象按照顺序经过各阶段
    Move objects through stages in sequence
  - 任意时刻，多个对象在同时被处理
    At any given times, multiple objects being processed

流水线原理

Principle of Pipeline

流水线原型
Prototype of Pipelines



Delay (延迟) = 320 ps

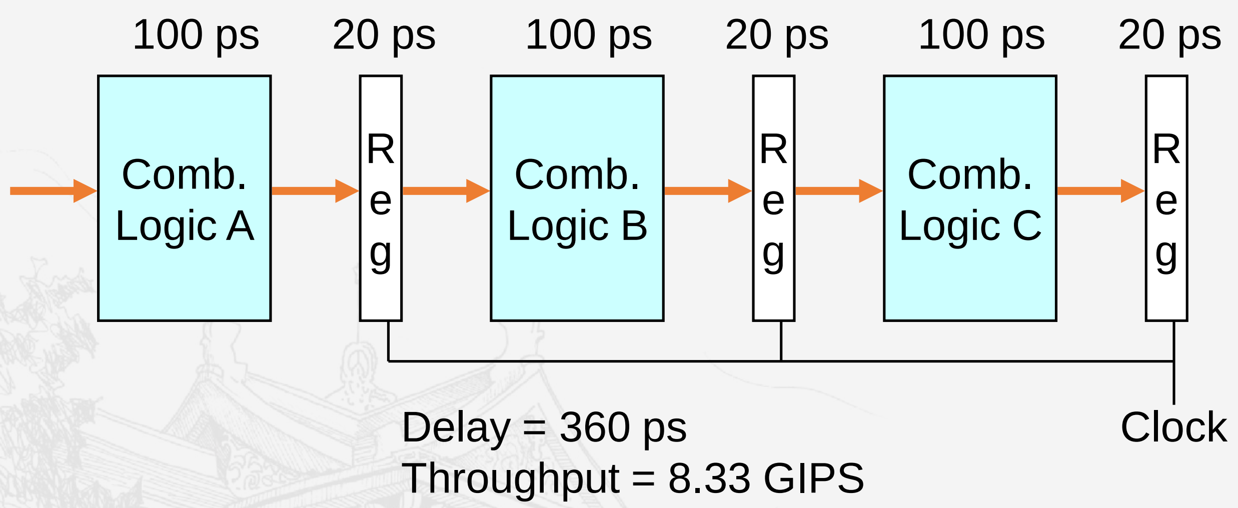
Throughput (吞吐量) = 3.12 GIPS

GIPS: Giga Instructions Per Second
十亿条指令/秒

流水线原理

Principle of Pipeline

三阶段流水线的版本
3-Way Pipelined Version



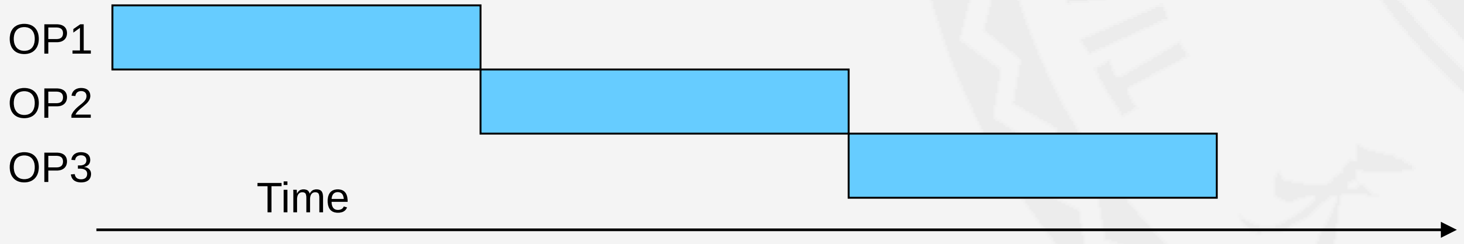
- 系统
  System
  - 将组合逻辑划分为三个阶段，每个阶段延迟 100 ps
    Divide combinational logic into 3 blocks of 100 ps each
  - 当阶段A的处理的前一个指令向后传递后，阶段A就可以开启下一条指令的处理
    Can begin new operation as soon as previous one passes through stage A.
    - 每120ps 开启一个新的处理
      Begin new operation every 120 ps
  - 总的延迟增加了：360ps
    Overall latency increases: 360ps

流水线原理

Principle of Pipeline

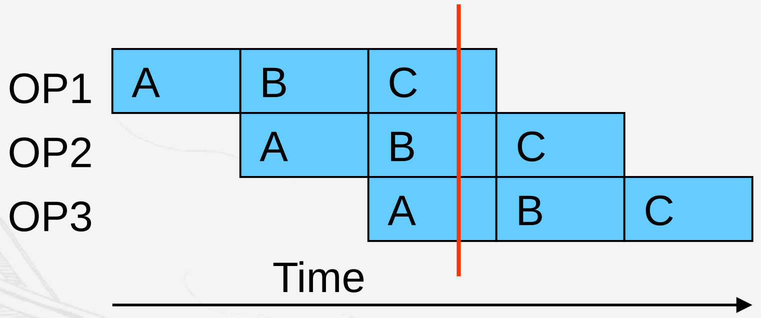
流水线示意图
Pipeline Diagrams

无流水线
Unpipelined



在前一条指令结束前不能开始新的指令
Cannot start new instruction until previous one completes

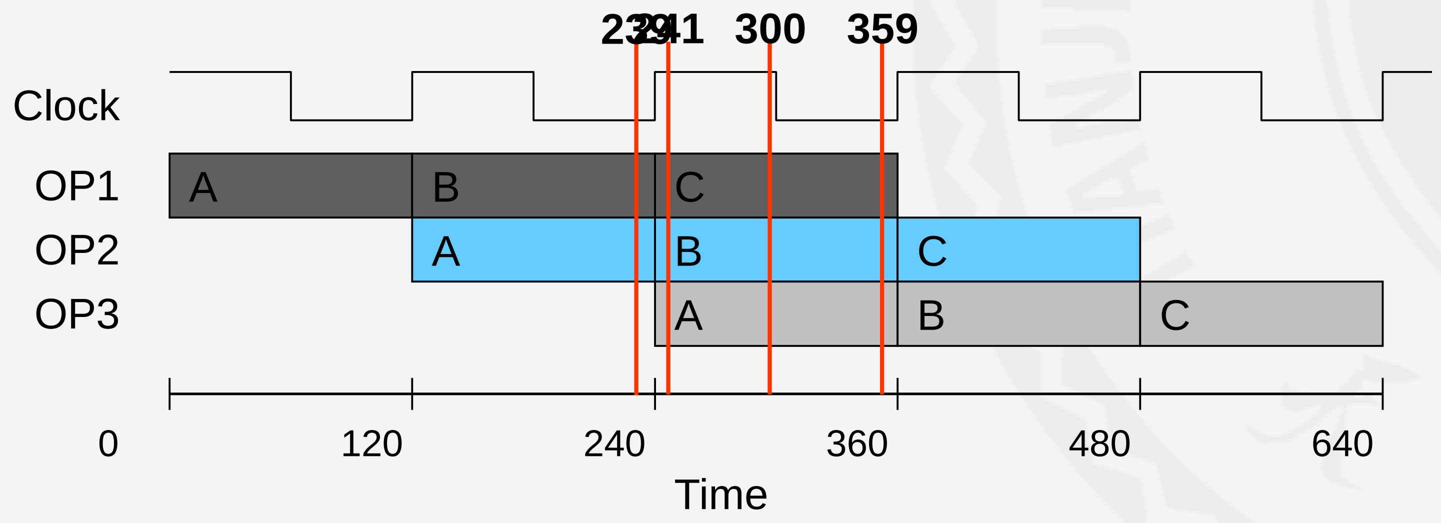
3阶段流水线
3-Way Pipelined

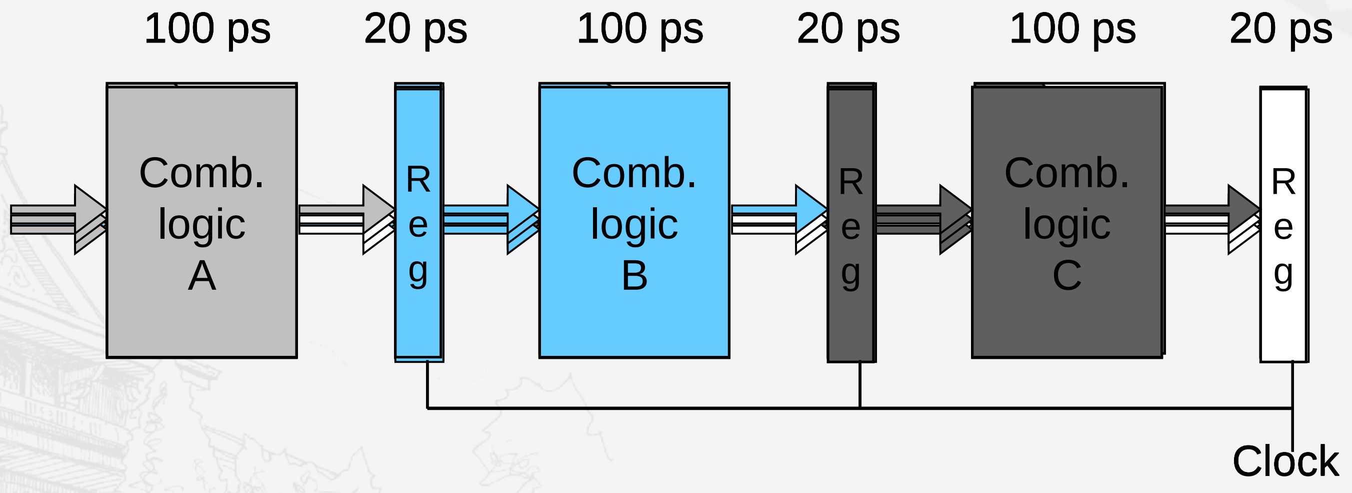


最多可以有三条指令同时执行
Up to 3 operations in process simultaneously

流水线原理

Principle of Pipeline

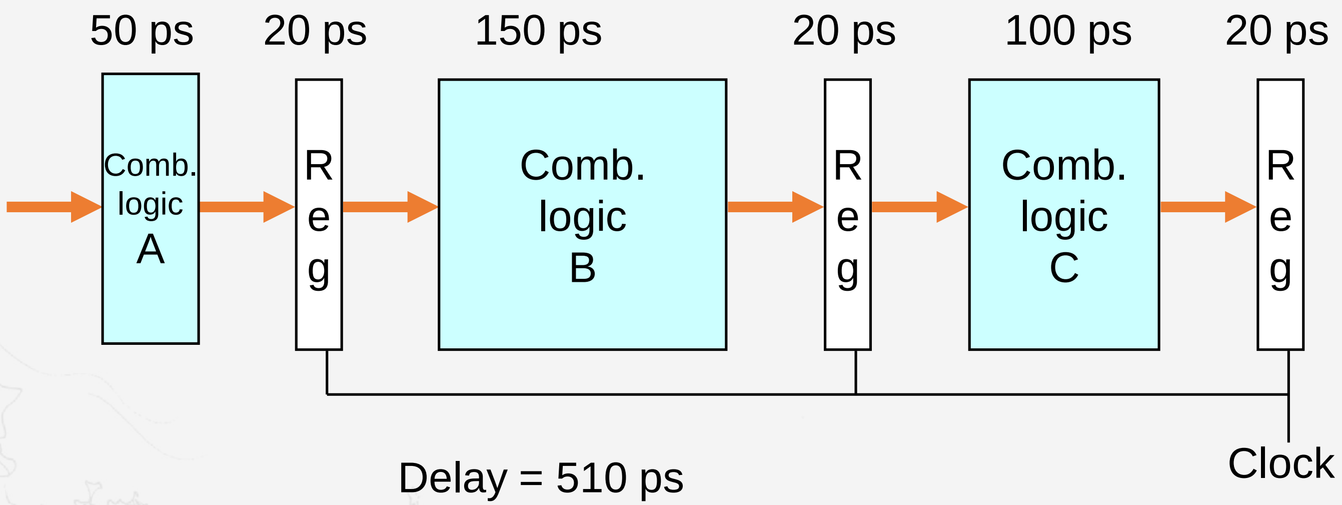




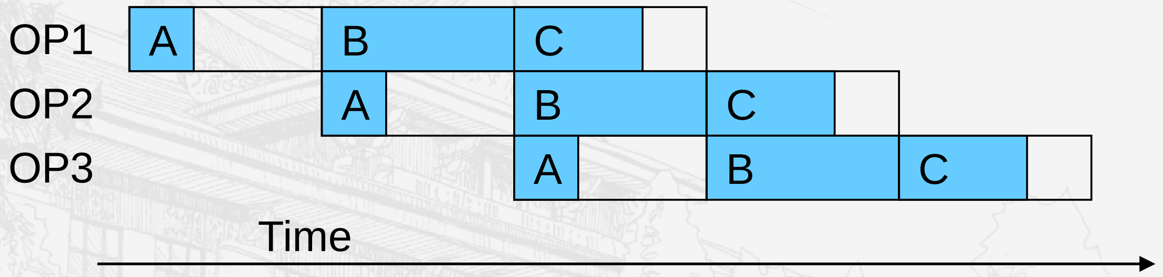
流水线原理

Principle of Pipeline

局限性：不一致的延迟
Limitations: Nonuniform Delays



Delay = 510 ps
Throughput = 5.88 GIPS

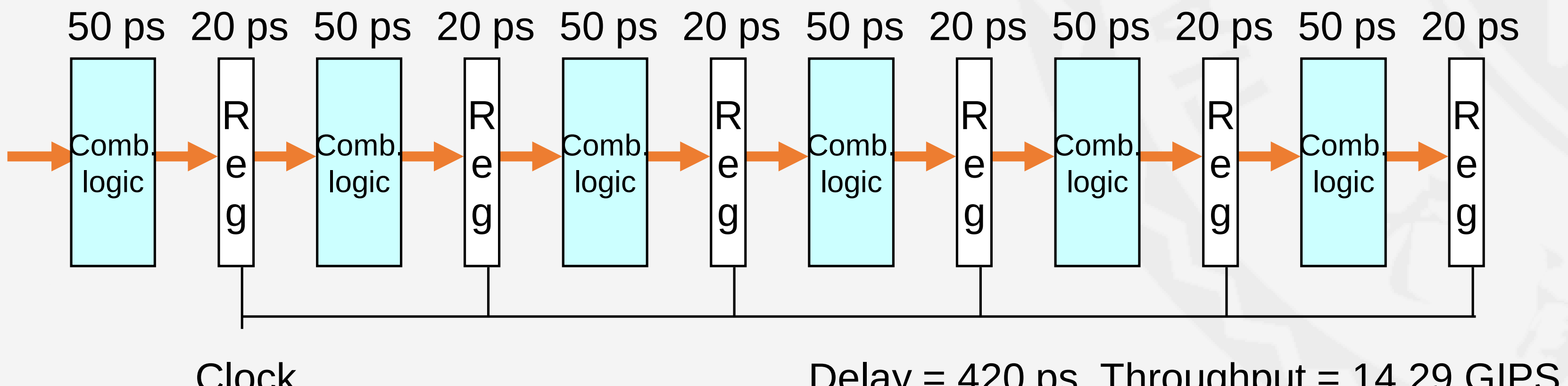


- （指令）吞吐量受限于最慢的阶段
  Throughput limited by slowest stage
- 其它阶段有较多时间处于空闲状态
  Other stages sit idle for much of the time
- 阶段均匀划分是一种设计上的挑战
  Challenging to partition system into balanced stages

流水线原理

Principle of Pipeline

局限性：寄存器的开销
Limitations: Nonuniform Delays



- 随着流水线的不断深入，总的寄存器的访问开销变得越来越显著
  As try to deepen pipeline, overhead of loading registers becomes more significant
- 寄存器访问开销在一个时钟周期中的占比
  Percentage of clock cycle spent loading register:

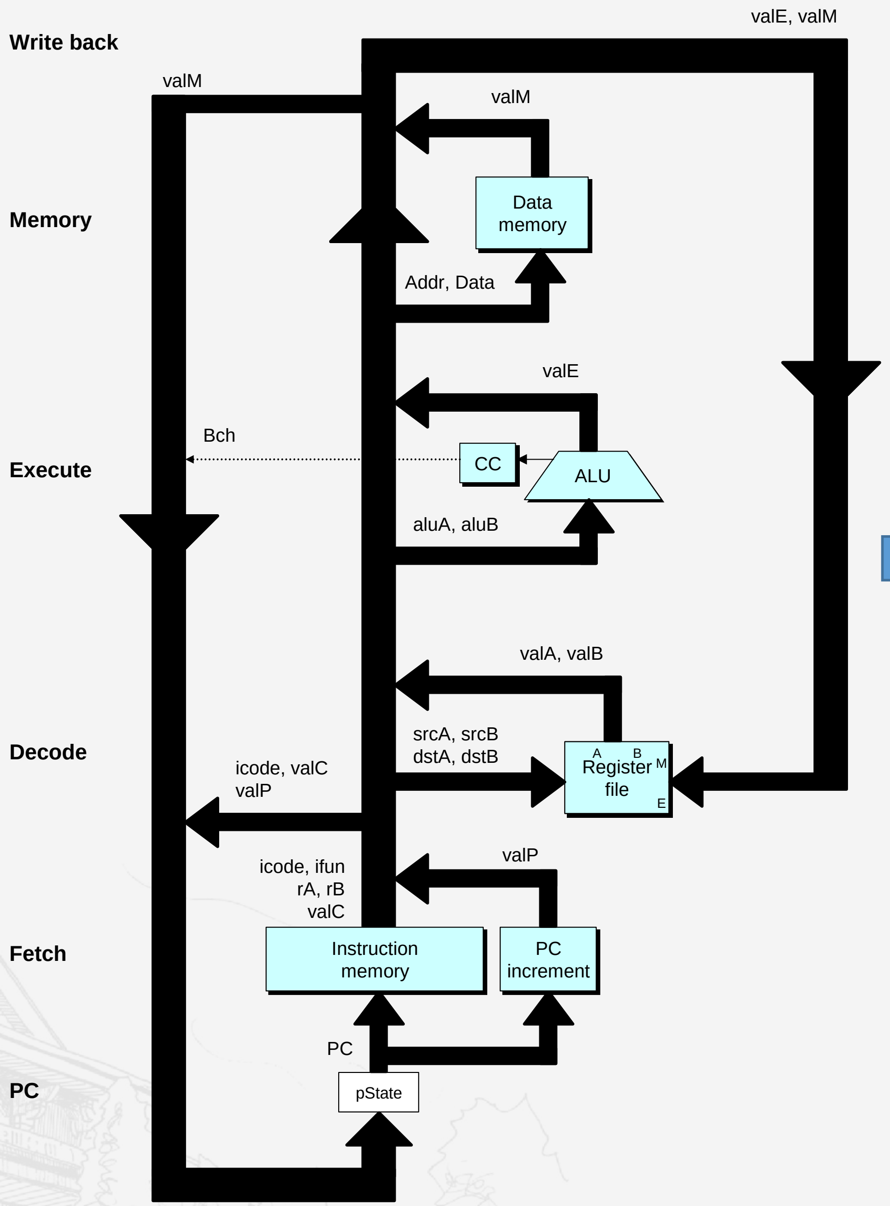
1-stage pipeline:	6.25%
3-stage pipeline:	16.67%
6-stage pipeline:	28.57%

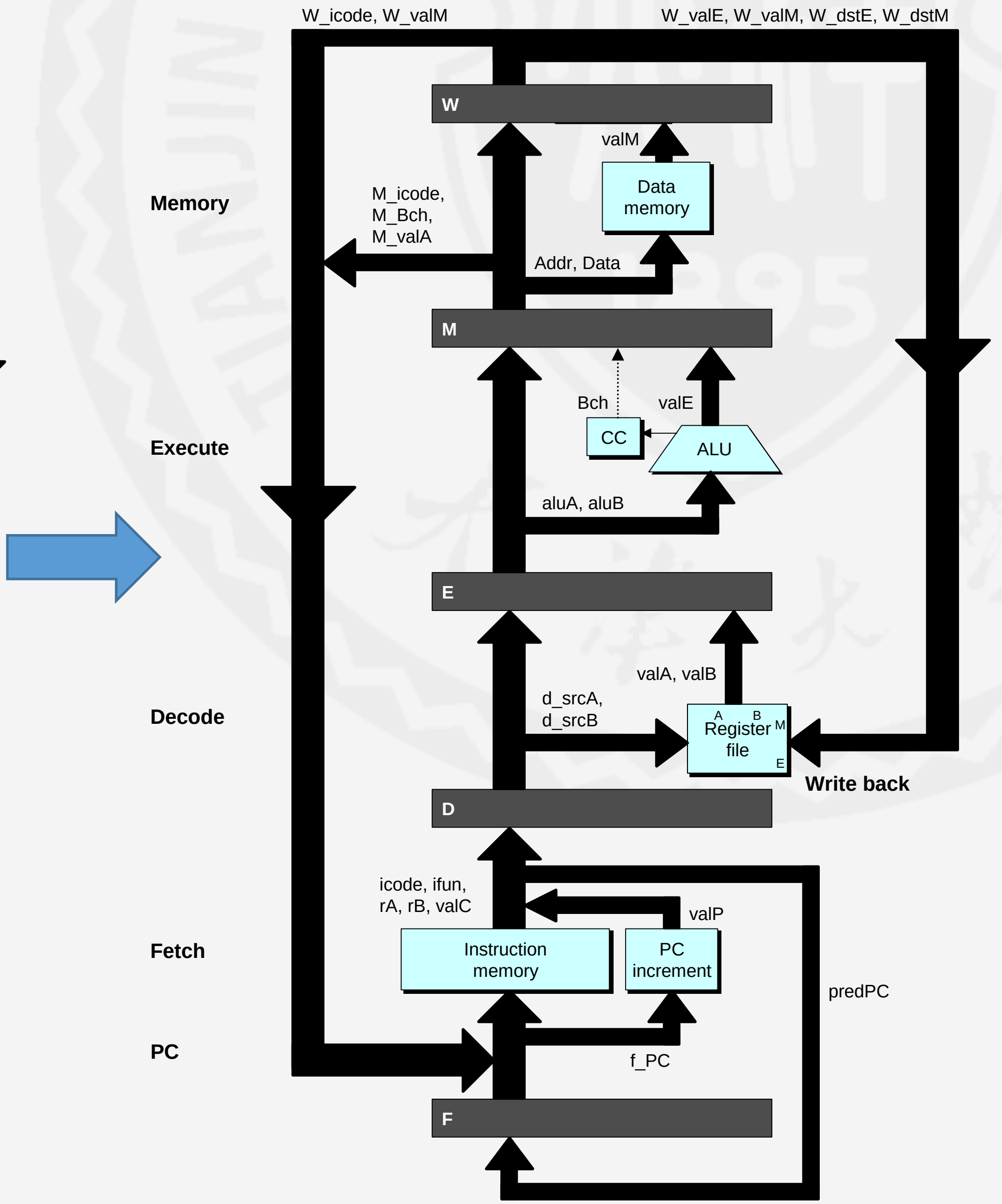
- 现代处理器的高性能是通过深度流水线实现的
  High speeds of modern processor designs obtained through very deep pipelining

流水线原理

Principle of Pipeline

插入流水线寄存器
Adding Pipeline Registers





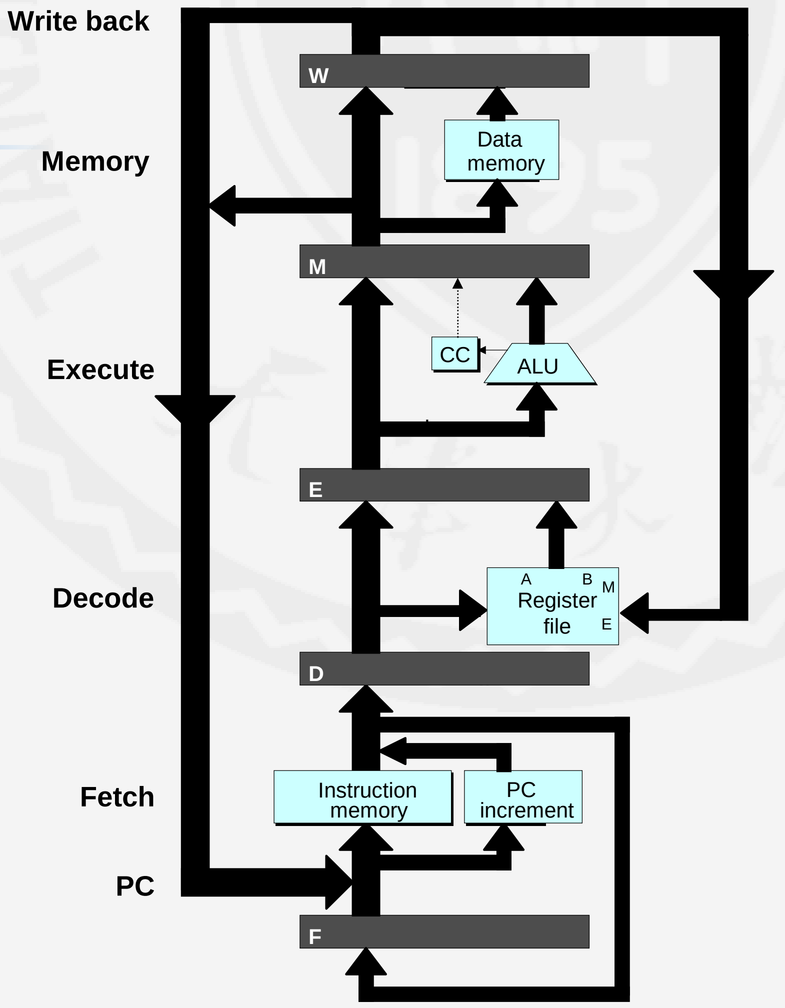
流水线原理

Principle of Pipeline

为了实现更加一致的划分，取指阶段包含了读指令和更新PC两个功能。

典型的5阶段流水线
Classic 5-Way Pipeline

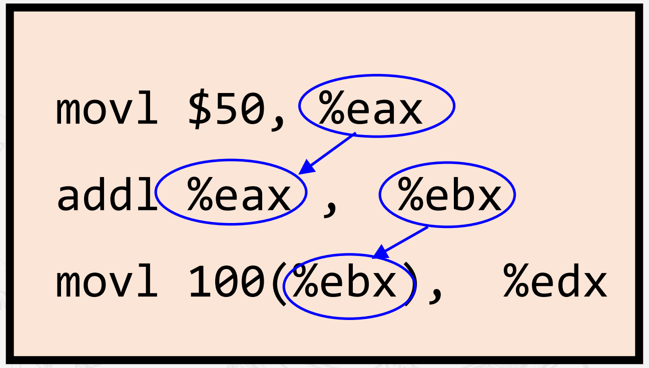
- 取指
  Fetch
  - 根据当前PC读取指令
    Read instruction according to current PC
  - 更新PC
    Compute incremented PC
- 译码
  Decode
  - 读寄存器
    Read program registers
- 执行
  Execute
  - 算术逻辑运算单元工作
    Operate ALU
- 访存
  Memory
  - 从内存读数据或向内存写数据
    Read or write data memory
- 写回
  Write Back
  - 更新寄存器
    Update register file



流水线原理

Principle of Pipeline

流水线中的数据相关性问题
Data Dependencies in Pipeline



```
movl $50, %eax
addl %eax , %ebx
movl 100(%ebx), %edx
```

- 一条指令的结果作为另一条指令的操作数
  Result from one instruction used as operand for another
  - 写后读（RAW）相关性
    Read-after-write (RAW) dependency
- 这种情况在实际的程序中非常常见
  Very common in actual programs
- 必须确保流水线可以妥善的处理这个问题
  Must make sure our pipeline handles these properly
  - 获得正确的计算结果
    Get correct results
  - 把对性能的影响降到最小
    Minimize performance impact

- 解决方案
  Solution
  - 暂停
    Stalling
  - 旁路
    Bypassing
  - 乱序执行
    out-of-order execution

流水线中的控制相关性问题
Control Dependencies in Pipeline

```
loop:
    subl %edx, %ebx
    jne target
    movl $10, %edx
    jmp loop
target:
    ret
```

- `jne`指令产生了一个控制相关

  The `jne` instruction create a control dependency

  - 接下来要执行哪一条指令?

    Which instruction will be executed?

解决方案：流水线中的控制相关性问题
Solution: Control Dependencies in Pipeline

- 动态分支预测
  Dynamic branch prediction
  - 随着程序行为的变化，预测的目标进行改变
    The prediction changes as program behavior changes
  - 分支预测是由硬件实现的
    Branch prediction implemented in hardware
  - 常见的分支预测算法都是基于分支的历史信息
    common algorithm based on branch history

- 静态分支预测
  Static branch prediction
  - 由编译器进行预测
    Compiler-determined prediction
  - gcc中的实现
    Implementation in gcc
    long __builtin_expect (long exp, long c)

流水线原理

Principle of Pipeline

```
//x is usually 1
#define likely(x)   __builtin_expect(!!(x), 1)
//x is usually 0
#define unlikely(x) __builtin_expect(!!(x), 0)

long test1(long a, long b) {
    if (likely(a>b))
        return 1;
    else
        return 0;
}

long test2(long a, long b) {
    if (unlikely(a>b))
        return 1;
    else
        return 0;
}
```

```
test1:
        cmpl    %rsi, %rdi
        jle .L3
        movl    $1, %rax
        ret
.L3:    movl    $0, %rax
        ret

test2:
        cmpl    %rsi, %rdi
        jg   .L7
        movl    $0, %rax
        ret
.L7:    movl    $1, %rax
        ret
```

流水线原理

Principle of Pipeline

重新审视条件数据移动指令
Conditional Move Revisited

```
long absdiff (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
absdiff:
   movq    %rdi, %rax   # x
   subq    %rsi, %rax   # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx   # eval = y-x
   cmpq    %rsi, %rdi   # x:y
   cmovle  %rdx, %rax   # if <=, result = eval
   ret
```

另一个分支预测的案例
Another Example: Branch Prediction

Looking at the Intel and AMD documentation for the rep instruction, we find that it is normally used to implement a repeating string operation. It seems completely inappropriate here.

The answer to this puzzle can be seen in AMD's guidelines to compiler writer. Their processors cannot properly predict the destination of a ret instruction when it is reached from a jump instruction.

The rep instruction serves as a form of no-operation here, and so inserting it as the jump destination does not change behavior of the code, except to make it faster on AMD processors

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

查阅Intel和AMD有关rep的文档，发现它通常是用来实现重复的字符串操作。这里用它似乎很不合适。

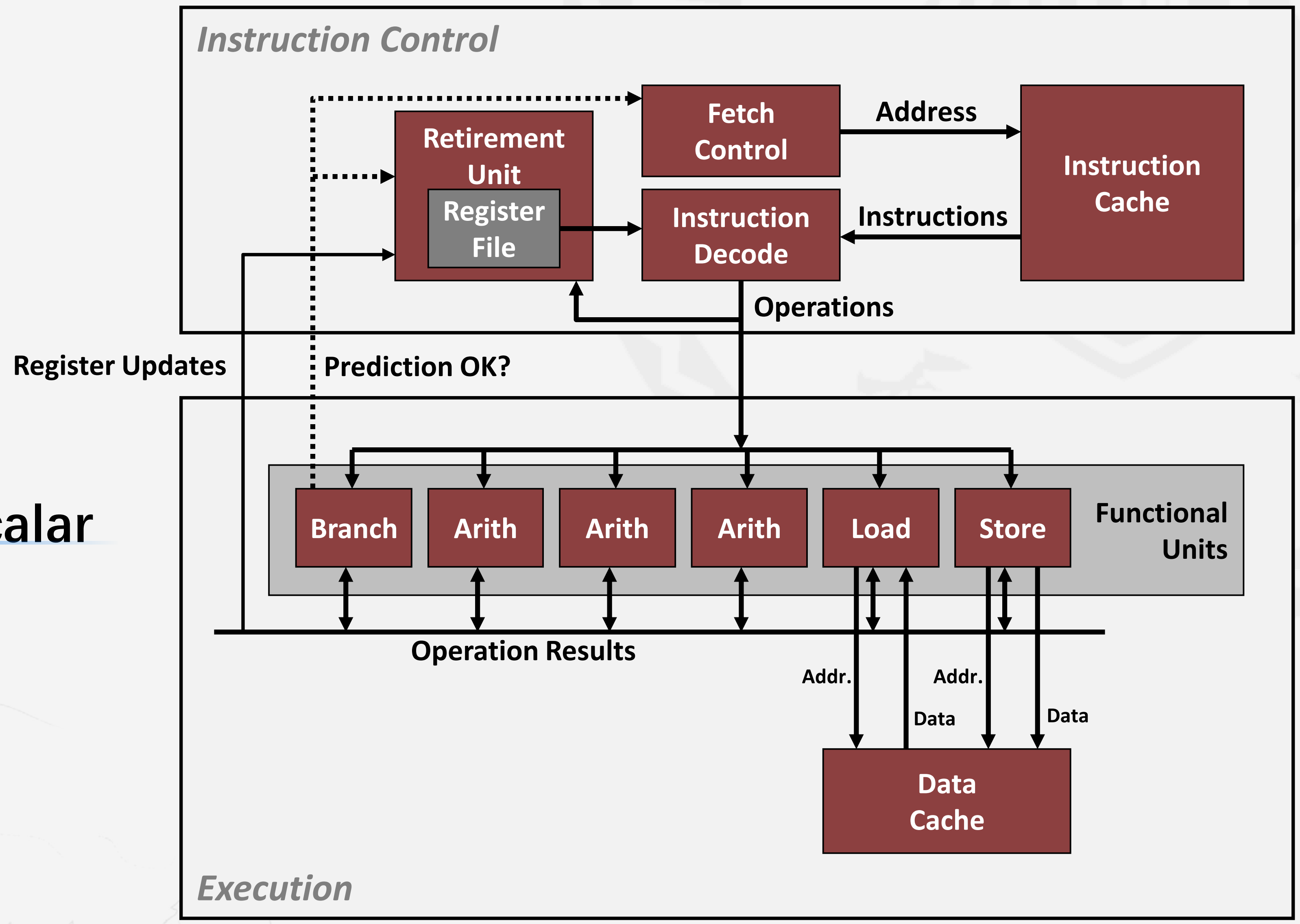
在AMD给编译器编写者的指导意见书中提到：当ret指令通过跳转指令到达时，处理器不能正确预测ret指令的目的。

这里的rep指令就是作为一种空操作，因此作为跳转目的插入它，能使代码在AMD上运行的更快，不会改变代码的其他行为。

最新趋势

state of the art

多发射和超标量
Multi-emission & Superscalar


Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instruction Decode
Instructions
Operations
Register Updates
Prediction OK?
Branch
Arith
Arith
Arith
Load
Store
Functional Units
Operation Results
Addr.
Addr.
Data
Data
Data Cache
Execution