体系结构Review

1 定量分析

2 01-03

相对性能

- 某程序运行在X系统上

$$performanc\ e(x) = \frac{1}{execution\ \_time(x)}$$

- X 性能是Y的n倍” 是指

- $$n = \frac{Performanc\ e(x)}{Performanc\ e(y)}$$

CPU 性能公式-CPI

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

- **注意：Instruction count 指的是（动态）执行的指令条数**

不同类型的指令具有不同的CPI

假设：CPI_i = 第i类指令执行所需的时钟周期数
IC_i = 所执行的第i类指令条数

$$\text{CPU cycles} = \sum_{i=1}^{n}(CPI_i \times IC_i)$$

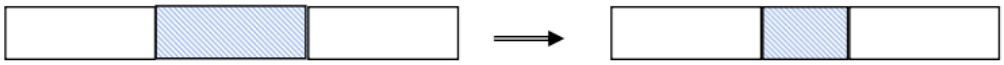
$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times IC_i)}{\sum_{i=1}^{n} IC_i} \qquad Feq_i = \frac{IC_i}{\sum_{i=1}^{n} IC_i}$$

$$\text{CPI} = \sum_{i=1}^{n}(CPI_i \times Feq_i)$$

Amdahl's Law

- 假设对构成机器的组件(或子系统）进行了改进（加速比的概念）

$$\text{Speedup(E)} = \frac{\text{ExTime (without E)}}{\text{ExTime (with E)}} = \frac{\text{Performance (with E)}}{\text{Performance (without E)}}$$



- 假设可改进部分E在原来的计算时间所占的比例为1-f，而部件加速比为S，任务的其他部分不受影响，则

$$\text{ExTime(with E)} = (f + (1-f)/S) \times \text{ExTime(without E)}$$
$$\text{Speedup(with E)} = 1/(f+(1-f)/S))$$

- **重要结论(收益递减)**：如果只针对整个任务的一部分进行优化，那么所获得的加速比不大于1/(1-F)

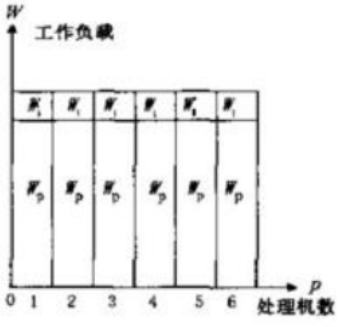
S为并行度或者CPU核数；Ws： 串行部分执行时间；

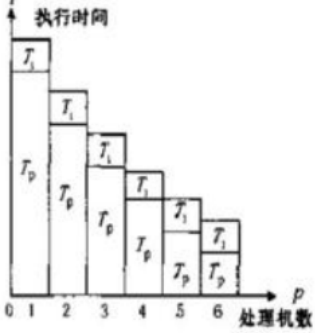
W： 串行执行总时间；p处理器数目； f： 串行执行的时间占比

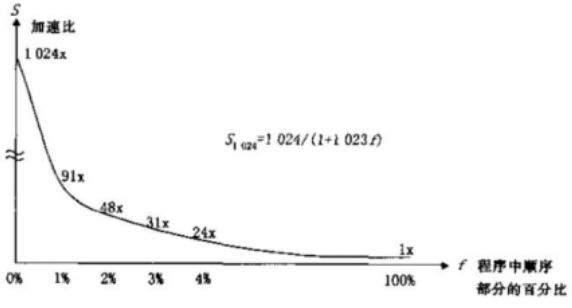
Amdahl定律

- **Amdahl 定律的假设**
  - 串行算法是给定问题的最优解决方案
    - 一些问题本质上是并行的，采用并行实现可大量减少所需计算步骤
  - 处理器核数增加，问题规模保持不变
- **Speedup = (Ws + (1-f)W)/(Ws+ (1-f)W/p)**

- **Amdahl 定律的几何意义**







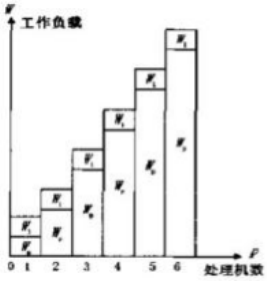
计算负载固定不变

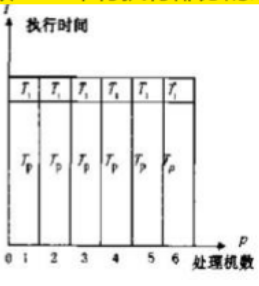
处理器数目增加，执行时间缩短

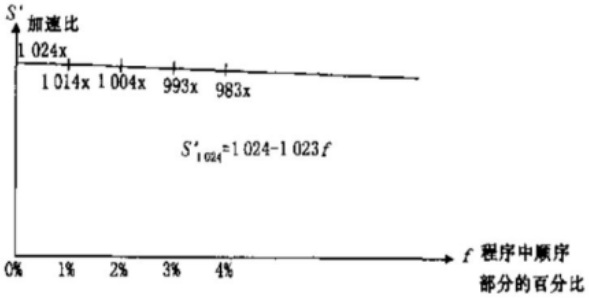
随着程序中串行执行占比提高,加速比下降

古斯塔夫森定律

- **有别于Amdahl定律的另一视角：扩展加速比（古斯塔夫森定律）**
  - 我们通常用更快的计算机解决更大规模的问题
  - 将处理时间视为常量，研究处理器数目的增加时，问题规模的增加情况（可扩放性）
    - 当问题规模增大时，程序的串行部分保持不变
    - 当增加处理器的数量时，每个处理器执行相同负载
  - Gustafson–Barsis’ s Law
- **Scaled Speedup = (Ws+(1-f)W×p) / (Ws + (1-f)W) = p-f(p-1)= (1-f)p+f**
  - **在串行比例保持不变的情况下，加速比与处理器数目p 基本成线性关系**
    - p：处理器核数 f：串行执行部分的时间占比







计算负载增加，为保证执行时间不变，需要增加处理器数

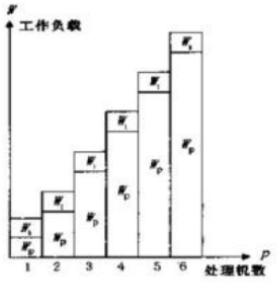
增加处理器数目，保证执行时间不变

随着程序中串行执行占比提高,加速比下降幅度减弱

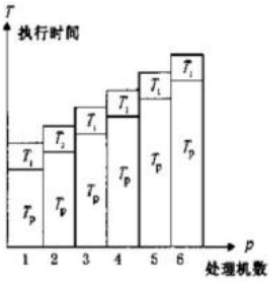
Sun and Ni's定理

- **假设：**
  - Ws：串行部分执行时间；W：串行执行总时间；p处理器数目，f：串行执行的时间占比
- **Amdahl定律**
  - Speedup = (Ws + (1-f)W)/(Ws+ (1-f)W/p)
- **Gustafson定律的几何意义**
  - Speedup = (Ws+(1-f)W×p) / (Ws + (1-f)W)
- **Sun and Ni's定律**
  - Amdahl和Gustafson定律的推广：考虑增大问题规模时，存储器访问开销会发生变化。
  - 引入了G(p)表示存储容量增加到p倍时，并行工作负载增加G(p)倍。加速比公式可表示为：

- **Speedup = (Ws+(1-f)G(p)W)/(Ws + (1-f)G(p)W/p)**
  - G(p) = 1：Amdahl 定律，工作负载无增加
  - G(p) = p：Gustafson定律



处理能力随处理器数目的增加C而增加



处理器增加，执行时间也随着增加

性能的综合评价

- **算术平均或加权的算术平均**
  - SUM(Ti)/n 或 SUM (Wi×Ti)/n

- **规格化执行时间，采用几何平均**

$$\sqrt[n]{\prod_{i=1}^{n} Execution\_time\_ratio_i}$$

  - SPEC采用这种方法(SPECRatio)

小结：定量分析基础

- **性能度量**
  - 响应时间 (response time)
  - 吞吐率 (Throughput)
- **CPU 执行时间 = IC × CPI × T**
  - CPI （Cycles per Instruction)
- **MIPS = Millions of Instructions Per Second**
- **Latency versus Bandwidth**
  - Latency指单个任务的执行时间，Bandwidth 指单位时间完成的任务量（rate)
  - Latency 的提升滞后于带宽的提升 (在过去的30年)
- **Amdahl定律用来度量加速比（speedup)**
  - 性能提升受限于任务中可加速部分所占的比例
- **Benchmarks：指一组用于测试的程序**
  - 比较计算机系统的性能
  - SPEC benchmark：针对一组应用综合性能值采用SPEC ratios 的几何平均

动态能耗和功耗

- **针对CMOS技术，动态的能量消耗是由于晶体管的on和off状态的切换引起的**
- **Dynamic Energy ∝ Capacitive Load × Voltage²**
  - the energy of pulse of the logic transition of 0-1-0 or 1-0-1
  - Capacitive Load = Capacitance of output transistors & wires
  - Voltage has dropped from 5V to below 1V in 20 years
- **Dynamic Power ∝**

  Capacitive Load × Voltage²× Frequency Switched
- **降低频率可以降低功耗**
- **降低频率导致执行时间增加 ->不能降低能耗**
- **降低电压可有效降低功耗和能耗**

动态功耗=电压^2×频率，能耗等于功耗×时间

TIPS：1、存储器访问与运算相比：能耗高2到4个数量级
2、DRAM访问不仅速度慢，而且能耗高
3、SRAM访问与DRAM访问相比，速度快，能耗低2个数量级
4、浮点运算与定点运算相比，能耗高，占用的面积大

静态功耗（Static Power）

- **当晶体管处于off状态时,漏电流产生的功耗称为静态功耗**
  - 随着晶体管尺寸的减少漏电流的大小在增加
- **Static Power = Static Current × Voltage**
  - Static power increases with the number of transistors
- **静态功耗有时会占到全部功耗的50%**
  - Large SRAM caches need static power to maintain their values
- **Power/Ground Gating: 通过切断不使用模块的电源或地减少漏电流**
  - To inactive modules to control the loss of leakage current

3 ISA

3.1 02-01

影响指令系统的主要因素

- **工艺技术的发展影响ISA的设计**
  - 早期硬件昂贵，简化硬件实现是ISA设计的主要任务
  - 随着半导体工艺技术的发展 →CISC
  - CPU速度与存储器速度的差异→Cache→**预取指令**
  - 工艺和功耗密度导致CPU主频极限→多核结构成为主流→**访存一致性和核间同步指令**
- **计算机体系结构**
  - ISA本身是体系结构的一部分，系统结构的变化直接影响ISA的设计
  - **例如：SIMD → 指令集扩展（AVX），多核结构直接影响着ISA的设计**
- **系统资源管理**
  - 多进程和虚拟地址空间：需要设计专门的地址翻译模块及配套的寄存器和指令
  - 操作系统所使用的**中断和异常需要ISA的支持**
  - 操作系统通常具有内核态和用户态，**需要设计专门的内核态指令**
  - **虚拟化技术，支持虚拟化的指令集扩展。如：Intel，AMD， ARM，**
- **编译技术**
  - RISC在某种意义上是在编译技术的推动的结果。RISC通常具有**16个以上通用寄存器**，以支持编译器有效地进行寄存器分配和调度指令
- **应用程序**
  - 指令是从各种算法中抽象出的公共算子。指令系统设计最终要为应用服务。

寻址方式小结

- **重要的寻址方式:** 统计基础上 → 充分利用指令长度
  - 偏移寻址方式，立即数寻址方式，寄存器间址方式
  - SPEC测试表明，使用频度达到 75%--99%
- **偏移字段的大小应该在 12 - 16 bits**
  - 可满足75%-99%的需求
- **立即数字段的大小应该在 8 -16 bits**
  - 可满足50%-80%的需求

指令格式选择策略

- **如果代码规模最重要，那么使用变长指令格式** → 代码紧凑度高
- **如果性能至关重要，使用固定长度指令**
- **有些嵌入式CPU附加可选模式，由每一应用自己选择性能还是代码量**
  - 窄指令支持更少的操作、更短的地址和立即数字段、更少的寄存器以及双地址格式，而不是传统的RISC计算机的三地址格式
  - 例如：RISC-V 的 RV32IC， C表示压缩表示
  - ARM Thumb ， microMIPS
- **有些机器使用边执行边解压的方式**
  - 例如 IBM 的CodePack PowerPC

ISA的功能设计

- **功能设计**
  - 任务：确定硬件支持哪些操作
  - 方法：统计的方法
  - **两种不同的设计理念**：CISC和RISC
- **CISC（Complex Instruction Set Computer）**
  - 目标：**强化指令功能**，减少运行的指令条数，**提高系统性能**
  - 方法：①面向目标程序的优化，②面向高级语言和编译器的优化
- **RISC（Reduced Instruction Set Computer）**
  - 目标：通过**简化指令系统**，用**高效**的方法**实现最常用的指令**
  - 方法：充分发挥流水线的效率，降低（优化）CPI

CISC计算机ISA的功能设计

$$CPUtime = IC \times CPI \times T$$

- **思路与目标：**
  - 强化指令功能，减少指令条数，以提高系统性能
- **基本优化方法**

1. 面向目标程序的优化是提高计算机系统性能最直接方法

  - 优化目标
    - 缩短程序的长度（Instruction Counts）
    - 缩短程序的执行时间
  - 优化方法
    - 统计分析**目标程序**执行情况，找出使用频度高，执行时间长的**指令或指令串**
    - 优化使用频度高的指令
    - 用新的指令代替使用频度高的指令串

优化目标程序的主要途径

1）增强运算型指令的功能

如sin(x), cos(x), SQRT(X)，甚至多项式计算

如用一条三地址指令完成

$P(X) = C(0)+C(1)X+C(2)X^2+C(3)X^3+.....$

2) 增强数据传送类指令的功能

主要是指数据块传送指令

R-R, R-M, M-M之间的数据块传送可有效的支持向量和矩阵运算，如IBM370

R-Stack之间设置数据块传送指令，能够在程序调用和程序中断时，快速保存和恢复程序现场，如 VAX-11

3) 增强程序控制指令的功能

在CISC中，一般均设置了多种程序控制指令。

RISC的定义和特点

- **RISC是一种计算机体系结构的设计思想，它不是一种产品。**
- **RISC是近代计算机体系结构发展史中的一个里程碑**
- **早期对RISC特点的描述**
  - **采用Load/Store结构：存储器访问与运算分离**
  - 大多数指令在单周期内完成
  - **硬布线控制逻辑**
  - 减少指令和寻址方式的种类
  - 固定的指令格式
  - **注重代码的优化**
- **从目前的发展看，RISC体系结构还应具有如下特点：**
  - 面向寄存器结构
  - 十分重视流水线的执行效率 - 尽量减少断流
  - 重视优化编译技术
- **减少指令平均执行周期数是RISC思想的精华**

RISC为什么会减少CPI

- **硬件方面：**
  - 硬布线控制逻辑（组合）
  - 减少指令和寻址方式的种类
  - 使用固定格式
  - 采用Load/Store
  - 指令执行过程中设置多级流水线
- **软件方面：十分强调优化编译的作用**

Maurice Wilkes 提出了微程序设计的概念来设计处理器的控制逻辑

小结

- **ISA的功能设计：**
  - 确定硬件支持哪些操作
  - 设计方法是统计的方法
- **存在CISC和RISC两种类型**
  - CISC（Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer）
    - 目标：通过**简化**指令系统，用最**高效**的方法**实现最常用的指令**
    - 手段：充分发挥流水线的效率，降低（优化）CPI

Data path 以及Control Unit

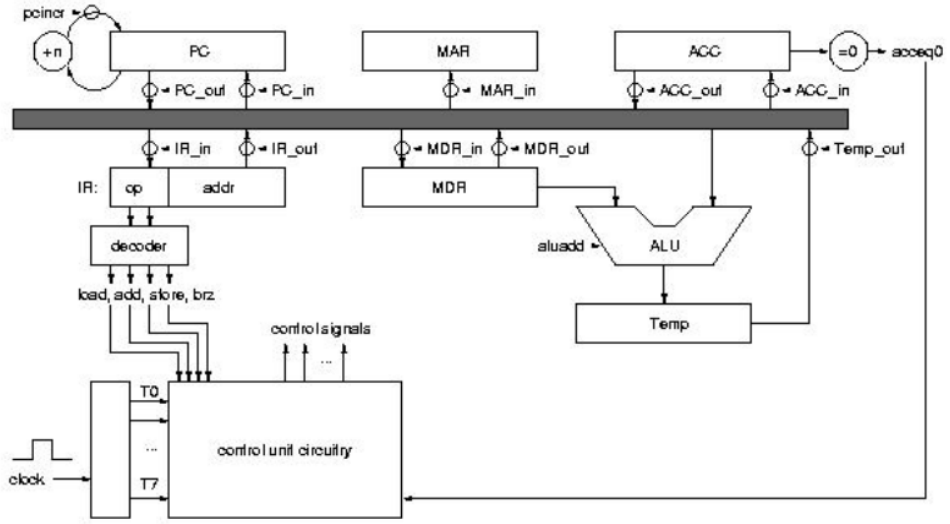


Figure 1. Simple data path for a four-instruction computer (the small circles represent control points)

(opcode 00) load address : ACC <- memory[address]
(opcode 01) add address : ACC <- ACC + memory[address]
(opcode 10) store address : memory[address] <- ACC
(opcode 11) brz address : if(ACC == 0) PC <- address

- **Data path、Control Unit**
- **Register Transfer**
- **Control Signal， Control Point (logic gate)**
- **Macroinstruction, Microinstruction, Microoperation**
- **组合逻辑控制器和微程序控制器**

A Simple Example

```
(opcode 00)  load  address :  ACC <- memory[ address ]
(opcode 01)  add   address :  ACC <- ACC + memory[ address ]
(opcode 10)  store address :  memory[ address ] <- ACC
(opcode 11)  brz   address :  if( ACC == 0 ) PC <- address
```

Figure 2：Instruction definitions for the simple computer

```
ACC_in    : ACC <- CPU internal bus
ACC_out   : CPU internal bus <- ACC
aluadd    : addition is selected as the ALU operation
IR_in     : IR <- CPU internal bus
IR_out    : CPU internal bus <- address portion of IR
MAR_in    : MAR <- CPU internal bus
MDR_in    : MDR <- CPU internal bus
MDR_out   : CPU internal bus <- MDR
PC_in     : PC <- CPU internal bus
PC_out    : CPU internal bus <- PC
pcincr    : PC <- PC + 1
read      : MDR <- memory[ MAR ]
TEMP_out  : CPU internal bus <- TEMP
write     : memory[ MAR ] <- MDR
```

Figure 3：Control signal definitions for the simple datapath

MAR = memory address

MDR = memory data register

```
time steps T0-T3 for each instruction fetch:
  T0: PC_out, MAR_in
  T1: read, pcincr
  T2: MDR_out, IR_in
  T3: time step (if needed) for decoding the opcode in the IR

time steps T4-T6 for the load instruction:
  T4: IR_out(addr part), MAR_in
  T5: read
  T6: MDR_out, ACC_in, reset to T0

time steps T4-T7 for the add instruction:
  T4: IR_out(addr part), MAR_in
  T5: read
  T6: ACC_out, aluadd
  T7: TEMP_out, ACC_in, reset to T0

time steps T4-T6 for the store instruction:
  T4: IR_out(addr part), MAR_in
  T5: ACC_out, MDR_in
  T6: write, reset to T0

time steps T4-T5 for the brz (branch on zero) instruction:
  T4: if (acceq0) then { IR_out(addr part), PC_in }
  T5: reset to T0
```

Figure 4. Control sequences for the four instructions

Microprogramming is far from extinct

- **80年代微程序控制起到了关键作用**
  - DEC uVAX, Motorola 68K series, Intel 286/386
- **现代微处理器中微程序控制扮演辅助的角色**
  - e.g., AMD Bulldozer, Intel Ivy Bridge, Intel Atom, IBM PowerPC, …
  - 大多数指令采用硬布线逻辑控制
  - 不常用的指令或者复杂的指令采用微程序控制
- **芯片bug的修复（打补丁） 例如Intel处理器在bootup阶段可装载微代码方式的patches**
  - 英特尔不得不重新启用微代码工具，并寻找原来的微代码工程师来修补**熔毁/幽灵安全漏洞**

技术目标

- **将ISA分成基础ISA和可选的扩展部分**
  - **ISA的基础部分足够简单、完整**，可以用于教学和嵌入式处理器，包括定制加速器的控制单元。它足够完整，可以运行软件栈。
  - **扩展部分提高计算的性能**，并支持多处理机并行
  - 支持32位和64位地址空间
- **方便根据应用需求扩展ISA（指令集扩展）**
  - 包括紧耦合功能单元和松耦合协处理器
- **支持变长指令集扩展**
  - 既为了提高代码密度，也为了扩展可能的自定义ISA扩展的空间
- **提供对现代标准的有效硬件支持**
- **用户级ISA和特权级ISA是正交的（相互独立，互不依赖）**
  - 在保持用户应用程序二进制接口(ABI)兼容性的同时，允许完全虚拟化，并允许在特权ISA中进行实验测试

RISC-V的指令编码

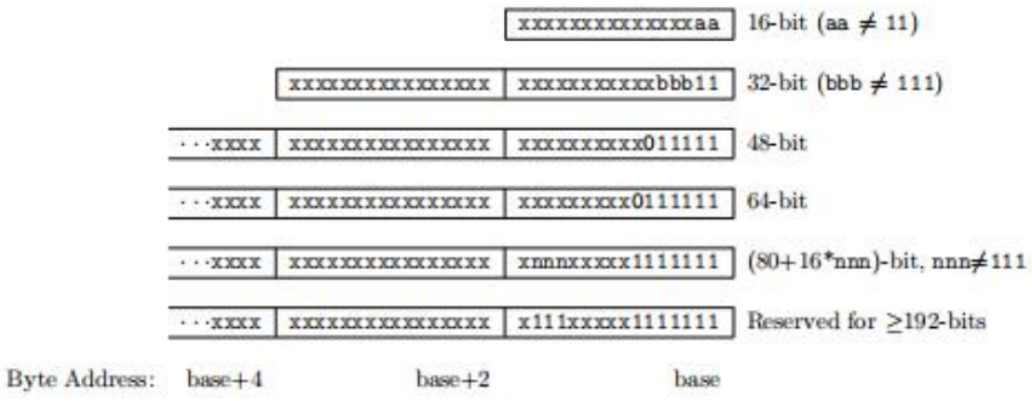
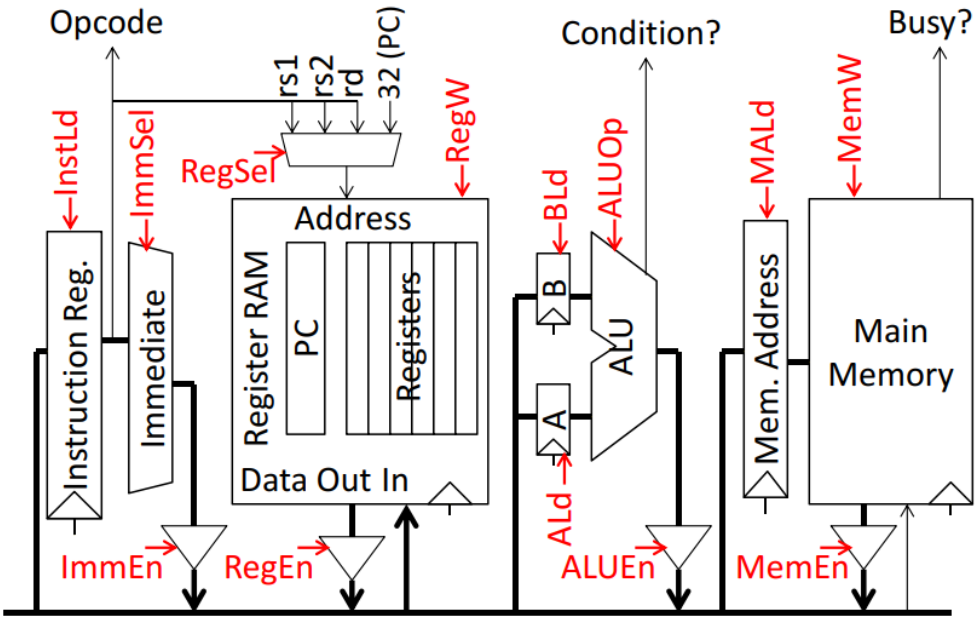


Figure 1.1: RISC-V instruction length encoding.

- **基本的32位整型类ISA指令长度32位，并且指令地址必须字对齐**
- **扩展的ISA支持变长指令格式，每条指令长度可以是2字节的倍数，指令地址按2字节对齐**

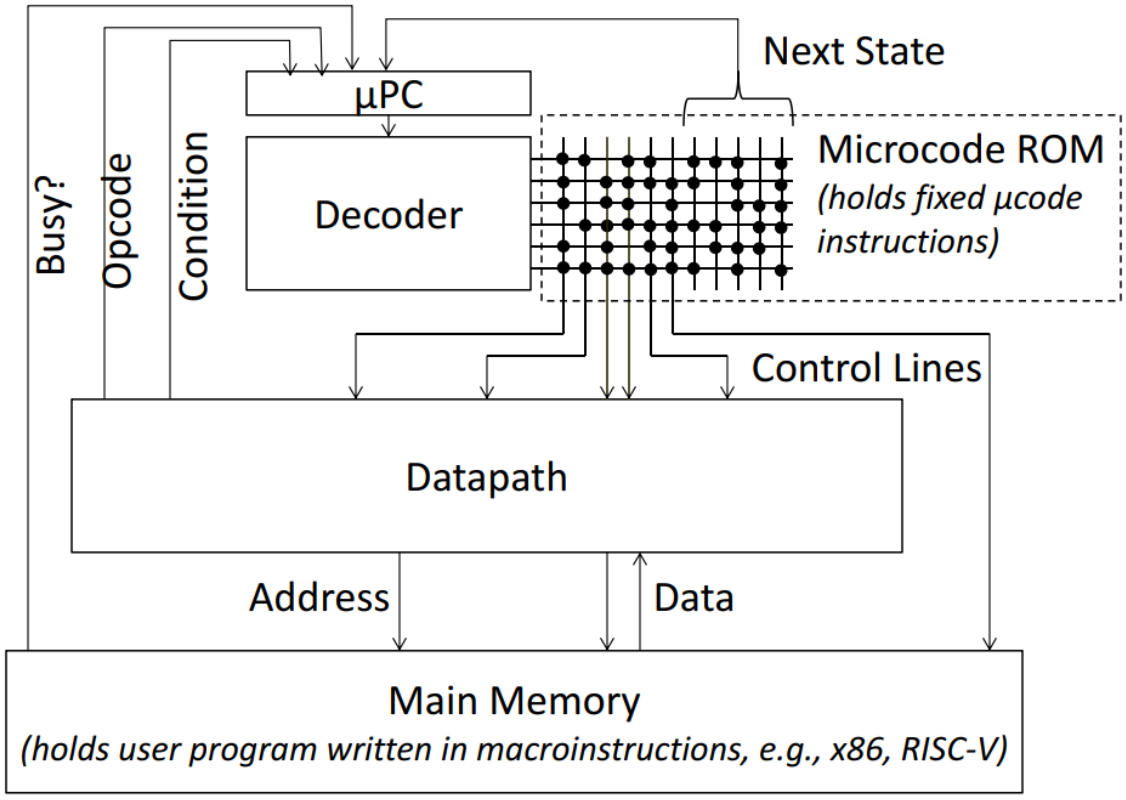
微程序控制RISC-V的单总线数据通路



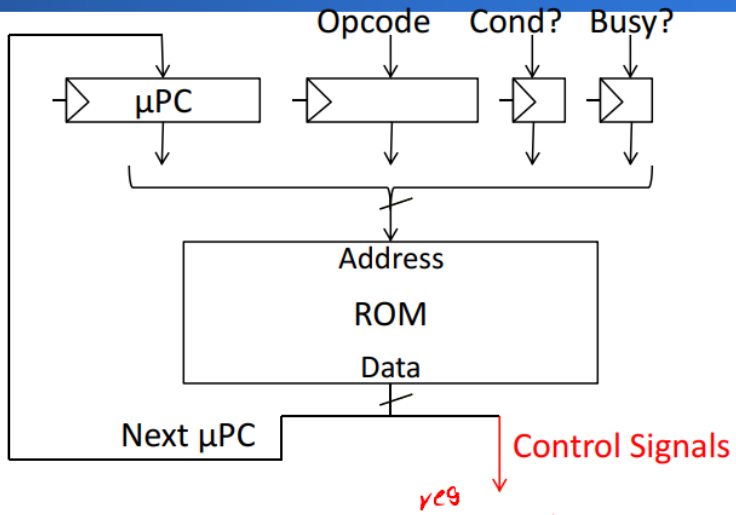
微指令的寄存器传输级(RTL)表示:

- **MA:=PC** means RegSel=PC; RegW=0; RegEn=1; MALd=1
- **B:=Reg[rs2]** means RegSel=rs2; RegW=0; RegEn=1; BLd=1
- **Reg[rd]:=A+B** means ALUop=Add; ALUEn=1; RegSel=rd; RegW=1

微程序控制 CPU



采用 ROM 实现微程序控制



- **How many address bits?**
  |μaddress| = |μPC|+|opcode|+ 1 + 1
- **How many data bits?**
  |data| = |μPC|+|control signals| = |μPC| + 18
- **Total ROM size = $2^{|\mu address|} \times |data|$**

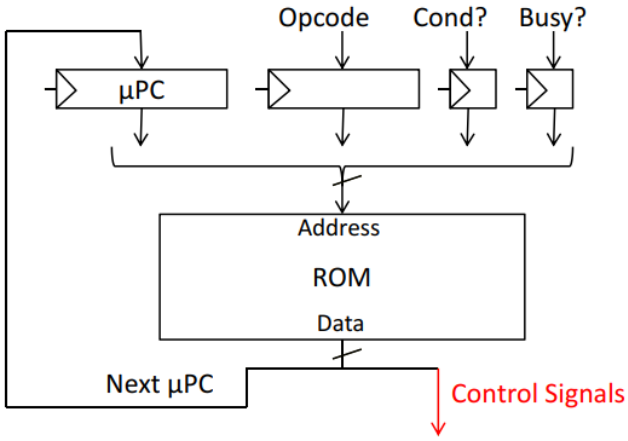
单总线数据通路结构的微程序控制存储器大小

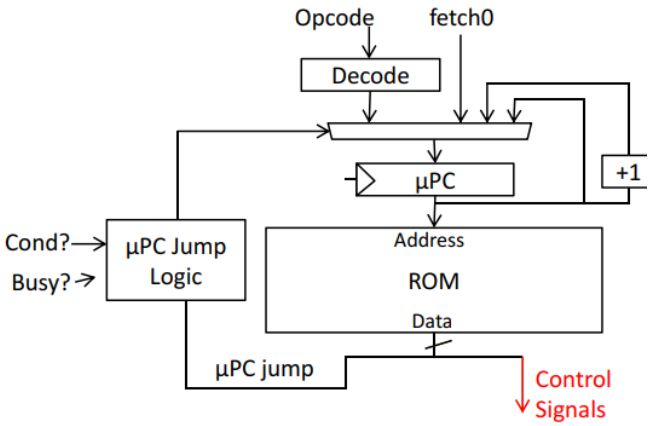
- **取指阶段有3个公操作**
- **RISC-V指令分为12 组**
- **完成一条指令需要5条微指令（包括dispatch）**
- **共计 3+12*5 = 63条微指令,因此 μPC需要6位**

- **指令操作码（Opcode)5 位, 每条微指令~18个控制信号**

- **微控制器的大小 = $2^{(6+5+2)} \times (6+18) = 2^{13} \times 24$ = ~25KiB!**

单总线 RISC-V 微程序控制引擎





µPC jump = next | spin | fetch | dispatch | ftrue | ffalse

|µaddress| = |µPC|+|opcode|+ 1 + 1

|data| = |µPC|+|control signals|

Total ROM size = $2^{|\mu address|} \times |data|$

Reducing Control Store Size

- **Reduce ROM height (#address bits)**
  - 使用外部逻辑来组合#input
  - 通过分组操作码减少#state
- **Reduce ROM width (#data bits)**
  - µPC 编码
  - 控制信号编码(vertical µcoding, nanocoding)