一. Fundamentals of Computer Design

一. Fundamentals of Computer Design

1.1 Introduction

1.2 Performance

1.3 Technology Trend

1.4 Quantitative approaches

1.4.1 CPU Performance

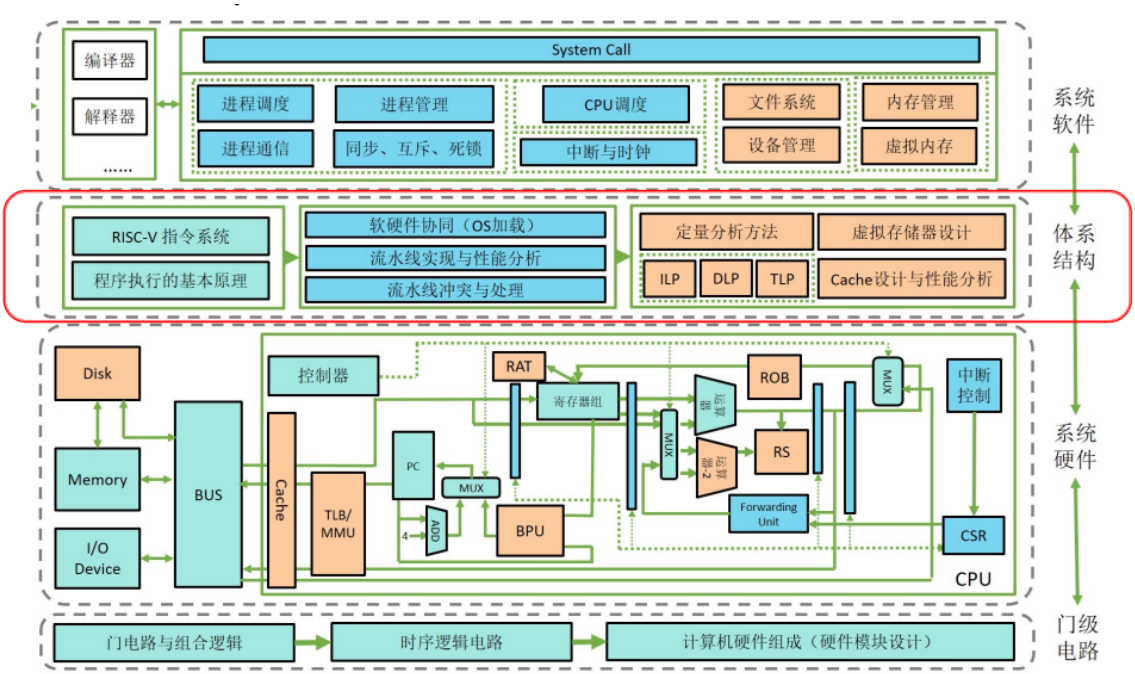
1.4.2 Amdahl's Law

1.5 Great Architecture Ideas

1.6 ISA

1.6.1 ISA Classification Basis

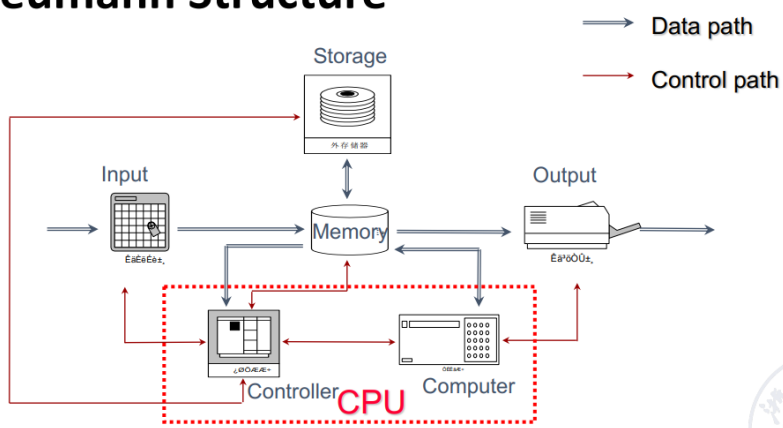
1.6.2 GPR（General-Propose Register） Classification



1.1 Introduction

Von Neumann Structure

Von Neumann Structure



Classes of Computers

- Desktop computers

  PC: Personal Computers

- Servers computers

  更强大的处理速度，容量（用于冗余备份）

- Embedded computers

  不能随意安装第三方应用的，与系统一体，称为嵌入式（不太符合国情 x

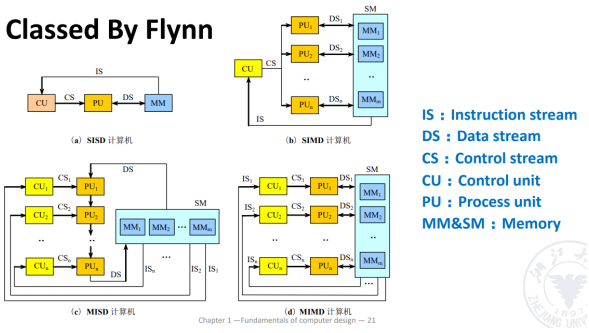
- Personal Mobile Devices

  如手机，iPad

- Supercomputer

Classed by Flynn

按照指令流和数据流进行分类



- SISD

  单指令流单数据流，如早期的单核 PC

- SIMD

  一条指令有多条数据流动（如向量数据），方便做流水线

- MISD

  多指令流单数据流，并不实际存在

- MIMD

  多指令流多数据流

Performance

- Alogrithm

- Programming language, compiler, architecture
- Processor and memory system
- I/O system (including OS)

> **Summary**
>
> **According to the process of using data, computers are developing in three fields:**
>
> - **speed up processing (parallel)**
> - **speed up transmission (accuracy)**
> - **Increase storage capacity and speed up storage (reliability)**

1.2 Performance

这里有很多因素会影响性能：体系结构，硬件实现，编译器，OS...

We need to be able to define a measure of performance.

- Single users on a PC -> a minimization of response time
- Large data -> a maximization of throughput

为了衡量性能，我们有响应时间和吞吐量两个指标：

- Latency (Response time 响应时间)

  一个事件开始到结束的时间
- Throughput (bandwidth 带宽)

  给定时间范围内完成了多少的工作量

这部分可见 **计组笔记**

The main goal of architecture improvement is to improve the performance of the system.

1.3 Technology Trend

The improvement of computer architecture

- Improvement of input / output
- The development of memory organization structure
- Two directions of instruction set development
    - CISC / RISC
- Parallel processing technology

  不同层次、粒度的并行

1.4 Quantitative approaches

1.4.1 CPU Performance

- CPU 执行时间 = CPU 时钟周期数 * CPU 时钟周期时间 = CPU 时钟周期数 / CPU 时钟频率
- - *IC：Instruction Count，指令数**
- - * `CPI：Cycle Per Instruction，每条指令的时钟周期数` **
    - 由 CPU 硬件决定
    - 不同的指令也会有不同的 CPI，平均 CPI 取决于指令的组合方式
    - CPI = CPU 时钟周期数 / IC
    - - * `CPU 执行时间 = IC * CPI / CPU 时钟频率` **

$$\begin{aligned}\text{CPU Execution time} &= \text{CPU Clock Cycles} \times \text{Clcok Period} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

👑 $$CPU\ Clock\ Cycles = Instructions\ for\ a\ Program \times Average\ Clock\ Cycles\ Per\ Instruction$$

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Period$$

$$CPU\ Time = \frac{Instruction\ Count \times CPI}{Clock\ Rate}$$

1.4.2 Amdahl's Law

Amdahl's Law: the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

当提升系统性能时，有多大的收益受限于被提升的部分所占的运行时间比例

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected}$$

Make the common case fast!

也被用来分析可行性

- 加速比

$$\begin{aligned}\text{Speedup} &= \frac{\text{Performance for entire task}_{\text{using Enhancement}}}{\text{Performance for entire task}_{\text{without Enhancement}}} \\ &= \frac{\text{Total Execution Time}_{\text{without Enhancement}}}{\text{Total Execution Time}_{\text{using Enhancement}}}\end{aligned}$$

`加速比 Sp = 改进后的性能 / 改进前的性能 = 改进前的时间 / 改进后的时间`

- 执行时间

$$\begin{aligned}T_{new} &= T_{old} \times (1 - fraction_{enhanced}) + T_{old} \times fraction_{enhanced} / speedup \\ &= T_{old} \times \left((1 - f) + \frac{f}{Sp} \right)\end{aligned}$$

f 指改进的部分所占的比例

- $$Sp_{overall} = \frac{T_{old}}{T_{new}} = \frac{1}{(1 - f) + \frac{f}{Sp}}$$

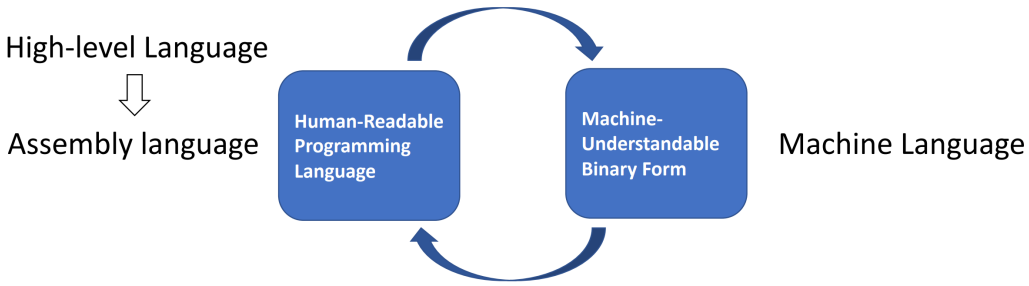
其中 Sp 为被优化部分的加速比，Sp_{overall} 为整体加速比，f 为被优化部分所占的运行时间比例

1.5 Great Architecture Ideas

- 摩尔定律
    - 每过 18-24 个月，集成电路的晶体管数量将增加一倍
- 使用抽象来简化设计
- 让最常见的情况更快
- 通过并行来提高性能
- 由很多级别的并行，比如指令集并行、进程并行等
- 通过流水线来提高性能
    - 将任务分为多段，让多个任务的不同阶段同时进行
    - 通常用来提高指令吞吐量
- 通过预测来提高性能
- 使用层次化的内存
    - 让最常访问的数据在更高层级，访问更快

1.6 ISA

- Instruction Set Architecture



Instruction Set Design Issues

- Where are operands stored?

  registers, memory, stack, accumulator
- How many explicit operands are there? (Classification of ISAs)

  0, 1, 2, or 3
- How is the operand location specified? (Addressing Modes)

  register, immediate, indirect, ...
- What type & size of operands are supported? (Data Representation)

  byte, int, float, double, string, vector, ...
- What operations are supported? (Types of Instructions)

  add, sub, mul, move, compare, ...

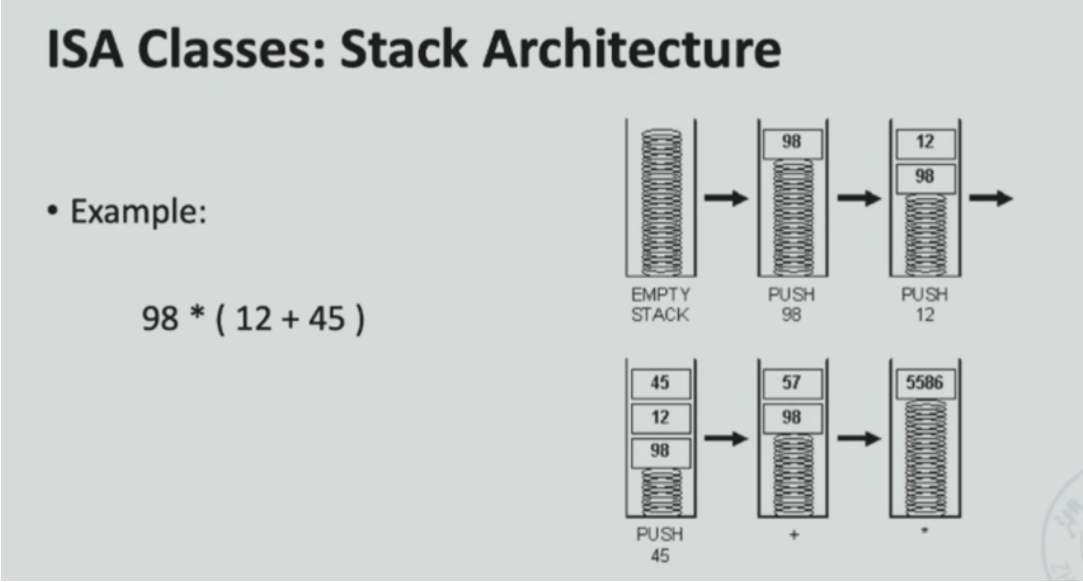
Basic Principles

- Compatibility
- Versatility
- High efficiency
- Security

1.6.1 ISA Classification Basis

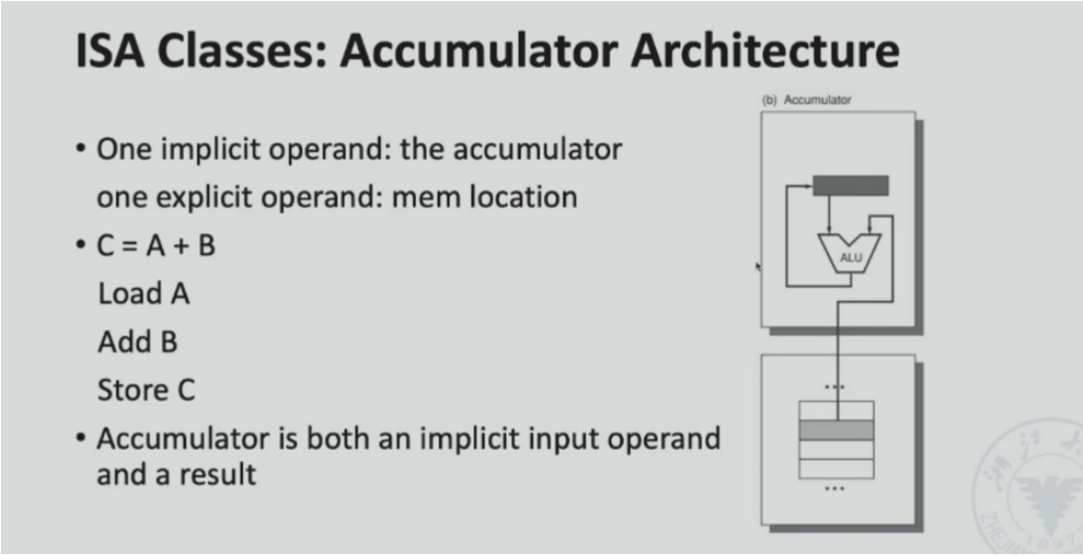
基本类型主要指的是从哪里取数，存到哪里以及计算的规则。

- stack
  First operand removed from second op replaced by the result.



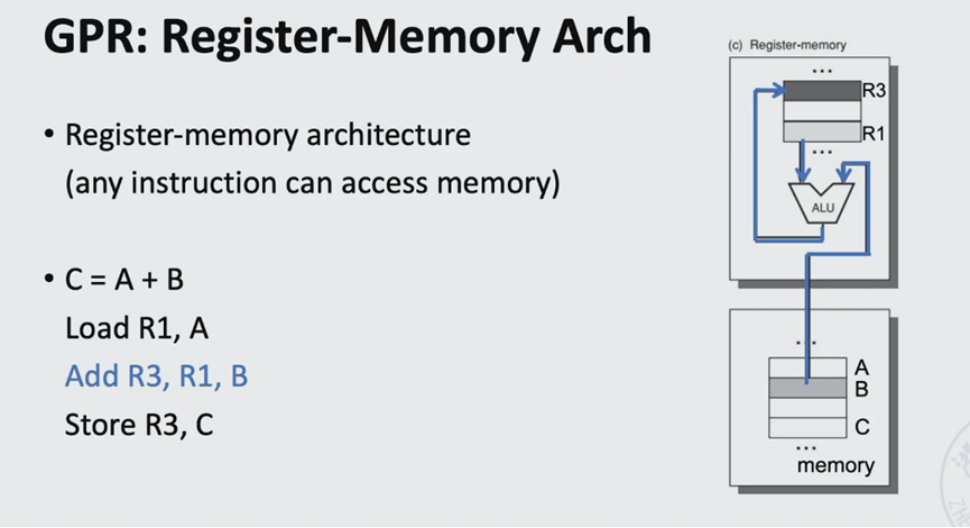
- accumulator
  - One implicit operand: the accumulator; one explicit operand: mem location
  - Accumulator is both an implicit input operand and a result

    累加器既是隐式输入操作数，又是结果



- register
  - Register-memory architecture

任何指令都可以访问内存

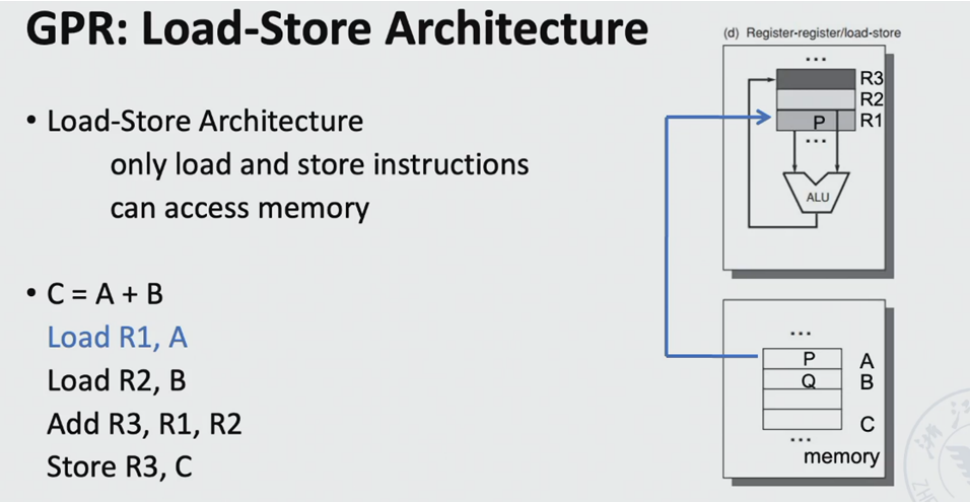


上述加法 A+B 的过程中，首先将 A Load 到寄存器 R1 中，然后由于 Add 指令也可以访存，因此再直接访问内存获取 B 的值，并与 R1 中 A 的值累加，结果存入寄存器 R3 中。

最后将结果 R3 存储到 C 当中。

- Load-store architecture

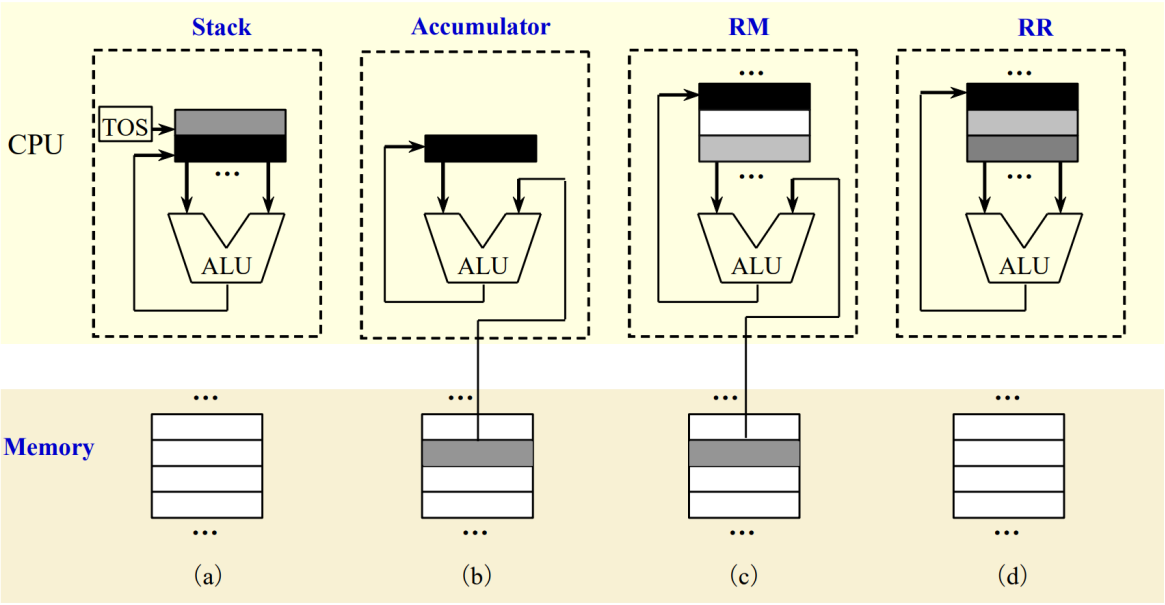
只有 load/store 的时候才能访存，其他时候都是基于寄存器操作



首先从内存中将两个操作数加载到寄存器中，然后运行 Add 指令，将结果写入 R3 写入 C 中（内存对应位置）。

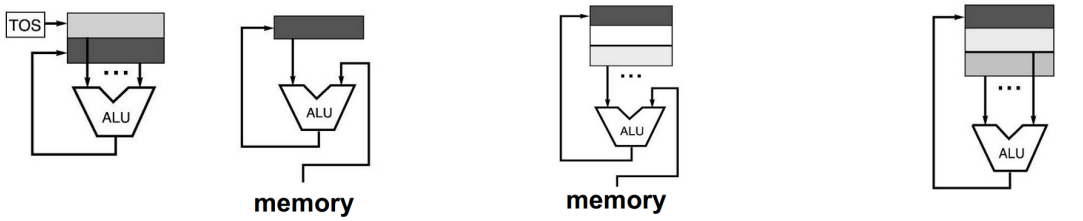
1.6.2 GPR（General-Propose Register） Classification

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)



!!! Example "A+B"

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1, A	Load R1,A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			Store C, R3



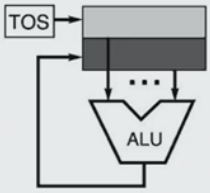
acc = acc + mem[C]

R1 = R1 + mem[C]

R3 = R1 + R2

More: try to do with $D = A * B - (A + C * B)$

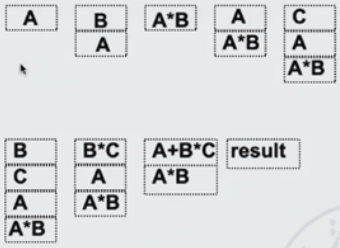
Stack Architectures



- Instruction set:
  - add, sub, mult, div, . . .
  - push A, pop A

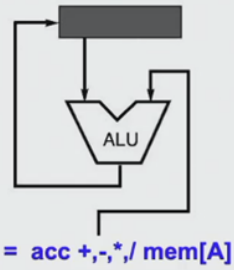
- Example: D = A * B - (A + C * B)

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



栈在计算时，始终计算的是栈顶的两个元素，然后计算结果推入栈中。

Accumulator Architectures



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

- Example: D = A * B - (A + C * B)

1. load B	5. load A
2. mul C	6. mul B
3. add A	7. sub D
4. store D	8. store D

累加器可以直接访问内存，只有一个操作数。

先将后一部分算好，即先算出 A+C*B，然后将得到的结果存入内存中。

然后再将前一部分算好，结果减去后一部分。

> Memory-Memory **中, 没有** Load **指令，每一条指令可以直接访存。**

- Example: D = A * B - (A + C * B)

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

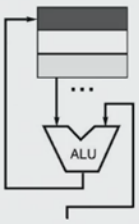
Memory-Memory 表示，两个源操作数都可以直接访问内存。优点是指令条数较少。

> Register-Memory 前一个操作数是寄存器，后一个操作数是直接访内存的。

Register-Memory Architectures

- Instruction set:

  add R1, A sub R1, A mul R1, B
  load R1, A store R1,



R1 = R1 +,-,*,/ mem[B]

- Example: D = A * B - (A + C * B)

```
1. load R1, A          5. mul R2, B    /* C*B */
2. mul R1, B  /* A*B */ 6. mul R2, B    /* C*B */
3. store R1, D         7. add R2, A    /* A + CB */
4. load R2, C          8. sub R2, D    /* AB - (A + C*B) */
                       9. store R2, D
```

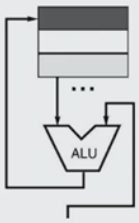
例如 add 指令中，只有一个操作数可以直接访问内存，另一个操作数必须从内存中 Load 到寄存器中。 因此，要先 Load R1, A，然后再 mul R1, B。

> Load-Store （Register-Register）只有 load/store 的时候才能访存，其他时候都是基于寄存器操作

Load-Store Architectures

- Instruction set:

  add R1, R2, R3 sub R1, R2, R3
  mul R1, R2, R3
  load R1, &A store R1, &A move R1, R2



R1 = R1 +,-,*,/ mem[B]

- Example: D = A * B - (A + C * B)

```
1. load R1, &A          5. add R8, R7, R1    /* A+C*B */
2. load R2, &B          6. mul R9, R1, R2    /* A*B */
3. load R3, &C          7. sub R10, R9, R8   /* A*B - (A+C*B) */
4. mul R7, R3, R2 /* C*B */ 8. store R10, D
```

GPR 速度快，但是 GPR 太多也会有资源的浪费和性能下降（如寻找对应的寄存器）