Chapter 2: Instructions: Language of the Computer

2.1. Introduction

2.2. Operations of the Computer Hardware

2.3. Operands of the Computer Hardware

Arquitetura e Organização de Computadores

Introdução: conjunto de instruções

- Revisão (vocês já sabem):
  - As palavras da linguagem dos computadores são as: _____
  - O vocabulário (conjunto das palavras) que os computadores entender é chamado de: _____
  - É a interface entre o hardware e o software!

Introdução: conjunto de instruções

- O que aprenderemos?
  - Um conjunto de instruções real
    - Na forma escrita por pessoas
    - Na forma lida pelo computador
  - Usando abordagem “top-down”

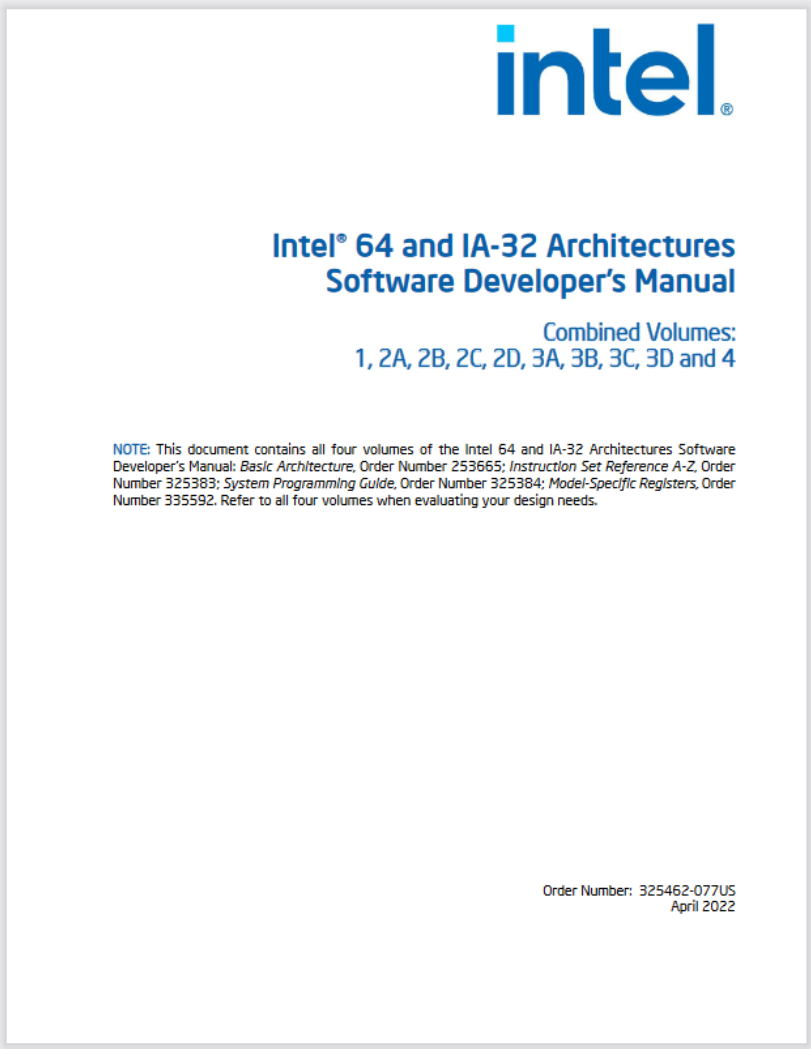
- Por quê?
  - Entender um dos “segredos” da computação, o conceito de **programa armazenado**:
    - Instruções e dados de diversos tipos podem ser armazenados na memória como números
  - Entender o impacto das linguagens de programação e otimização de compiladores na performance.

Introdução: conjunto de instruções

- Afinal: o que é a Arquitetura do Conjunto de Instruções?
  - É um **modelo abstrato de um computador** que define:
    - as instruções suportadas
    - os tipos de dados suportados
    - os registradores
    - o hardware para gerenciar a memória principal
    - características fundamentais (consistência de memória, modos de endereçamento, memória virtual, etc.)
    - modelo de entrada e saída (input e output)
  - Um dispositivo que executa as instruções definidas pelo ISA, como a CPU por exemplo, é chamado de **implementação**.
  - ATENÇÃO: o ISA especifica o **comportamento** do código de máquina rodando em implementações dessa ISA, de forma **independente das características da implementação**. Isso fornece **compatibilidade entre implementações diferentes**.

Introdução: conjunto de instruções

- É um **modelo abstrato de um computador**:



intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

NOTE: This document contains all four volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325462-077US
April 2022

4.834 páginas...
Vai cair tudo na prova!

"Resumão" com as principais instruções:
https://www.felixcloutier.com/x86/

Introdução: conjunto de instruções

- **Implementações diferentes do mesmo ISA**
  Intel x86-64 (EMT64, x64, AMD64, INTEL 64):

Introdução: conjunto de instruções

- Existem diversas **Arquiteturas de Conjuntos de Instruções (ISA)** que podem ser categorizadas, grosso modo, nos seguintes tipos:
  - **CISC**: Complex Instruction Set Computer
    - É melhor fazer de tudo um pouco: > 1000 instruções
    - Faz “trocentas” coisas
  - **RISC**: Reduced Instruction Set Computer
    - É melhor fazer tudo de pouco: +- 200 instruções
    - Limita cada instrução a (quase sempre):
      - Acessar 3 registradores
      - Acessar 1 memória
      - Realizar 1 operação
  - LIW: Long Instruction Word
  - VLIW: Very Long Instruction Word
  - EPIC: Explicitly Parallel Instruction Computer
  - Outras (teóricas):
    - MISC: Minimal Instruction Set Computer
    - OISC: One-Instruction Set Computer

Introdução: conjunto de instruções

- Se existem tantos ISAs diferentes, como aprender? Babel?
  - CISC:
    - Intel x86-64
    - PDP-11
    - VAX
    - ...
  - RISC:
    - IBM Power PC
    - Sun Sparc
    - MIPS
    - Alpha
    - RISC-V
    - ARM
    - ...

Pieter Bruegel the Elder - The Tower of Babel (Vienna) - 1563

Introdução: conjunto de instruções

- Os ISAs são mais ou menos como dialetos da mesma língua! Aprendeu um, sabe o resto.
  - CISC:
    - Intel x86-64
    - PDP-11
    - VAX
    - ...
  - RISC:
    - IBM Power PC
    - Sun Sparc
    - MIPS
    - Alpha
    - RISC-V
    - ARM
    - ...

Introdução: conjunto de instruções

- Qual ISA vamos aprender? O **RISC-V**! Por quê?
  - Desenvolvido na Universidade de Berkeley, em 2010
  - Padrão aberto e gratuito
  - Comunidade internacional
  - Produtos sendo lançados
  - Popularidade crescente

Introdução: conjunto de instruções

The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191213

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu
December 13, 2019

The RISC-V Instruction Set Manual
Volume II: Privileged Architecture
Document Version 20211203

Editors: Andrew Waterman[1], Krste Asanović[1,2], John Hauser
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu, jh.riscv@jhauser.us
December 4, 2021

Introdução: conjunto de instruções

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR ReTired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

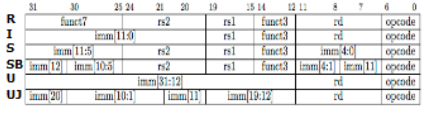
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

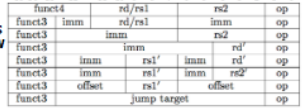
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)	+RV{64,128}
Multiply	MULtiply	R	MUL rd,rs1,rs2	MUL{W|D} rd,rs1,rs2
	MULtiply upper Half	R	MULH rd,rs1,rs2	
	MULtiply Half Sign/Uns	R	MULHSU rd,rs1,rs2	
	MULtiply upper Half Uns	R	MULHU rd,rs1,rs2	
Divide	DIVide	R	DIV rd,rs1,rs2	DIV{W|D} rd,rs1,rs2
	DIVide Unsigned	R	DIVU rd,rs1,rs2	
Remainder	REMainder	R	REM rd,rs1,rs2	REM{W|D} rd,rs1,rs2
	REMainder Unsigned	R	REMU rd,rs1,rs2	REMU{W|D} rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)	+RV{64,128}
Load	Load Reserved	R	LR.W rd,rs1	LR.{D|Q} rd,rs1
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.{D|Q} rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.{D|Q} rd,rs1,rs2
Add	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.{D|Q} rd,rs1,rs2
Logical	XOR	R	AMOXOR.W rd,rs1,rs2	AMOXOR.{D|Q} rd,rs1,rs2
	AND	R	AMOAND.W rd,rs1,rs2	AMOAND.{D|Q} rd,rs1,rs2
	OR	R	AMOOR.W rd,rs1,rs2	AMOOR.{D|O} rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W rd,rs1,rs2	AMOMIN.{D|Q} rd,rs1,rs2
	MAXimum	R	AMOMAX.W rd,rs1,rs2	AMOMAX.{D|Q} rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W rd,rs1,rs2	AMOMINU.{D|Q} rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W rd,rs1,rs2	AMOMAXU.{D|Q} rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)	+RV{64,128}
Move	Move from Integer	R	FMV.{H|S}.X rd,rs1	FMV.{D|Q}.X rd,rs1
	Move to Integer	R	FMV.X.{H|S} rd,rs1	FMV.X.{D|Q} rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W rd,rs1	FCVT.{H|S|D|Q}.{L|T} rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU rd,rs1	FCVT.{H|S|D|Q}.{L|T}U rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q} rd,rs1	FCVT.{L|T}.{H|S|D|Q} rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q} rd,rs1	FCVT.{L|T}U.{H|S|D|Q} rd,rs1
Load	Load	I	FL{W,D,Q} rd,rs1,imm	
Store	Store	S	FS{W,D,Q} rs1,rs2,imm	
Arithmetic	ADD	R	FADD.{S|D|Q} rd,rs1,rs2	
	SUBtract	R	FSUB.{S|D|Q} rd,rs1,rs2	
	MULtiply	R	FMUL.{S|D|Q} rd,rs1,rs2	
	DIVide	R	FDIV.{S|D|Q} rd,rs1,rs2	
	SQuare RooT	R	FSQRT.{S|D|Q} rd,rs1	
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q} rd,rs1,rs2,rs3	
	Multiply-SUBtract	R	FMSUB.{S|D|Q} rd,rs1,rs2,rs3	
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q} rd,rs1,rs2,rs3	
	Negative Multiply-ADD	R	FNMADD.{S|D|Q} rd,rs1,rs2,rs3	
Sign Inject	SIGN source	R	FSGNJ.{S|D|Q} rd,rs1,rs2	
	Negative SIGN source	R	FSGNJN.{S|D|Q} rd,rs1,rs2	
	Xor SIGN source	R	FSGNJX.{S|D|Q} rd,rs1,rs2	
Min/Max	MINimum	R	FMIN.{S|D|Q} rd,rs1,rs2	
	MAXimum	R	FMAX.{S|D|Q} rd,rs1,rs2	
Compare	Compare Float =	R	FEQ.{S|D|Q} rd,rs1,rs2	
	Compare Float <	R	FLT.{S|D|Q} rd,rs1,rs2	
	Compare Float ≤	R	FLE.{S|D|Q} rd,rs1,rs2	
Categorization	Classify Type	R	FCLASS.{S|D|Q} rd,rs1	
Configuration	Read Status	R	FRCSR rd	
	Read Rounding Mode	R	FRRM rd	
	Read Flags	R	FRFLAGS rd	
	Swap Status Reg	R	FSCSR rd,rs1	
	Swap Rounding Mode	R	FSRM rd,rs1	
	Swap Flags	R	FSFLAGS rd,rs1	
	Swap Rounding Mode Imm	I	FSRMI rd,imm	
	Swap Flags Imm	I	FSFLAGSI rd,imm	

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, { } means set, so L{D|Q} is both LD and LQ. See risc.org. (8/21/15 revision)

Introdução: conjunto de instruções

- Antes de continuarmos: por que você acha que os ISAs são mais como dialetos de uma linguagem e não como linguagens totalmente diferentes?

Introdução: conjunto de instruções

- Antes de continuarmos: por que você acha que os ISAs são mais como dialetos de uma linguagem e não como linguagens totalmente diferentes?
  - Tecnologia tem princípios fundamentais parecidos
  - Só existem poucas operações básicas (+, -, etc.)
  - Facilitar a construção do hardware e do compilador:
    - Simplificar o equipamento!
      - Idéia conhecida desde 1946!

Introdução: conjunto de instruções

```c
#include <stdio.h>

int main(void)
{
    printf("%s\n", "Olá, mundo!");
    return 0;
}
```

RISC-V rv32gc gcc 12.1.0 | Compiler options...

A | Output... | Filter... | Libraries | + Add new... | Add tool...

```
1   .LC0:
2           .string "Ol\303\241, mundo!"
3   main:
4           addi    sp,sp,-16
5           sw      ra,12(sp)
6           sw      s0,8(sp)
7           addi    s0,sp,16
8           lui     a5,%hi(.LC0)
9           addi    a0,a5,%lo(.LC0)
10          call    puts
11          li      a5,0
12          mv      a0,a5
13          lw      ra,12(sp)
14          lw      s0,8(sp)
15          addi    sp,sp,16
16          jr      ra
```

Operações: adição e subtração

- Toda **instrução aritmética** no RISC-V realiza **uma, e apenas uma operação**, e deve envolver exatamente **3 "variáveis"**
  - **2 operandos fontes: input/fonte de dados**
  - **1 operando destino: onde o resultado será armazenado**

  Exemplo: **a = b + c;**

```
add a, b, c
```

- Sua vez: como fazer, em RISC-V, a operação em C:
  a = b + c + d + e;

- Comentários são iniciados por **//**

- Cada linha contém, no máximo, 1 instrução.

Operações: adição e subtração

- Por que toda instrução aritmética exige sempre 3 variáveis?
- Por que toda instrução só realiza 1 única operação?

Operações: adição e subtração

- Por que toda instrução aritmética exige sempre 3 variáveis?
- Por que toda instrução só realiza 1 única operação?

1º Princípio de Projeto:
A Simplicidade Favorece a Regularidade

- Regularidade torna a implementação mais simples (hardware para número variável de operandos é mais complexo do que para um número fixo)
- Regularidade favorece aumentar performance com custo menor.

Operações: adição e subtração

- E a subtração? Mesma coisa...
  - **2 operandos fontes: input/fonte de dados**
  - **1 operando destino: onde o resultado será armazenado**

  Exemplo: **d = a – e;**

```
sub d, a, e
```

- Sua vez: como fazer, em RISC-V, a operação em C:
  f = (g + h) – (i + j);

Operandos: registradores!

- Considere o que você fez (eu espero!):
  f = (g + h) – (i + j);

```
add t0,  g, h
add t1,  i, j
sub  f, t0, t1
```

- O que são essas “variáveis”? Onde seus valores estão armazenados?

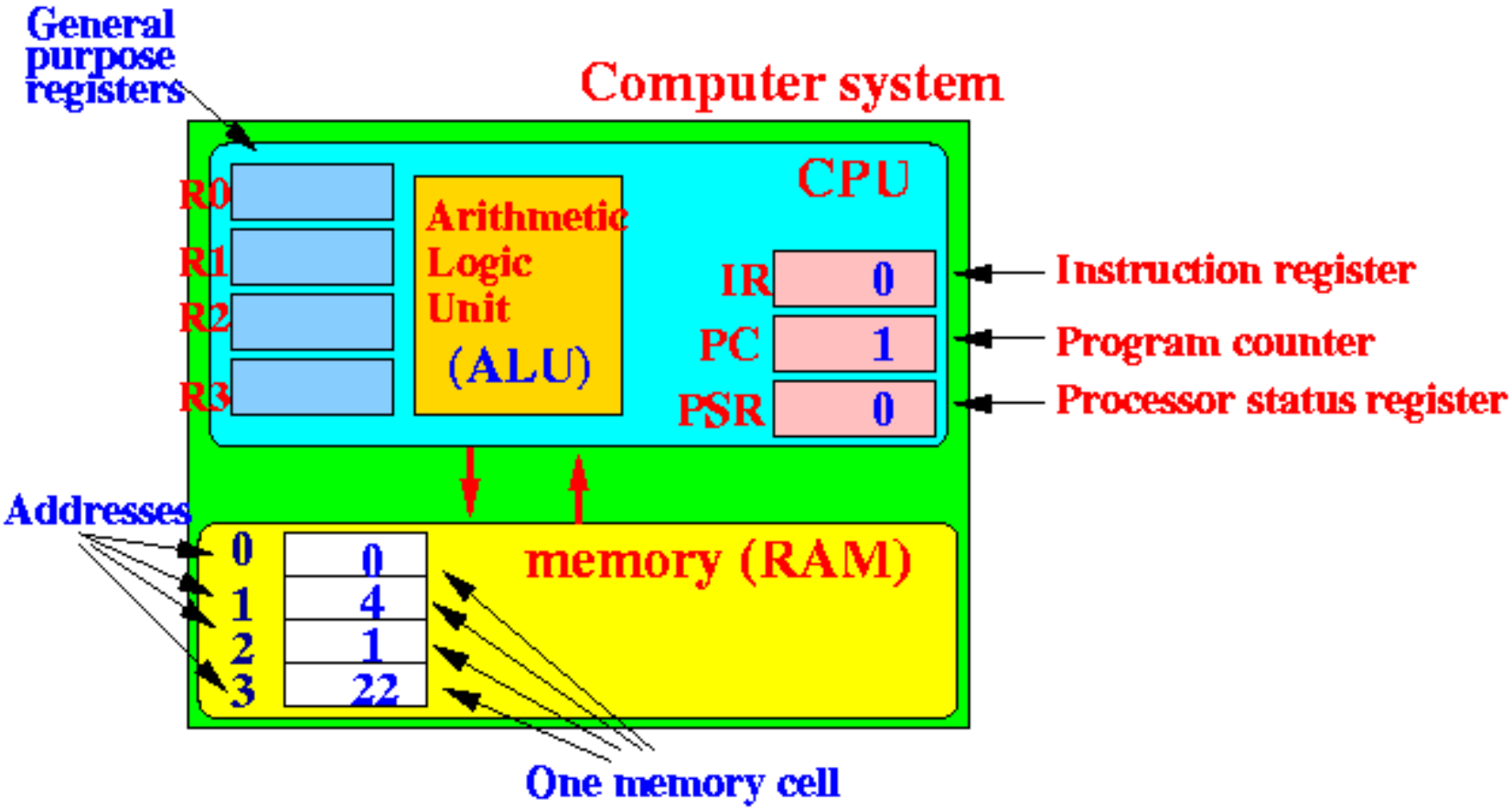
  – São **REGISTRADORES**, pequenas quantidades de memória de armazenamento **super-ultra-mega-hiper-rápidas** que existem diretamente **dentro da CPU**

Operandos: registradores!

- Já falei que registradores são rápidos?

Operandos: registradores!

- Já falei que registradores ficam dentro da CPU?



Operandos: registradores!

- Existem diversos “tipos” de registradores, e cada ISA especifica isso de modo diferente.

- No RISC-V existem **32 registradores de propósito geral**, para dados acessados com freqüência, cada um deles com tamanho de **32 bits**. Existem outros registradores específicos (serão vistos depois).

- Como o tamanho de cada registrador é 32 bits, a **palavra** (**word**) do RISC-V é de 32 bits.
  - Existe também a palavra dupla (doubleword), com 64 bits.

- A **palavra** (**word**) é a unidade de tamanho que a CPU consegue acessar de cada vez.

Operandos: registradores!

- Por que só 32 registradores? Considerando que praticamente não há limitação no espaço físico para alocar registradores dentro da CPU, por que não colocar 256 registradores na CPU e ter muito mais dados para trabalhar?

Operandos: registradores!

- Por que só 32 registradores? Considerando que praticamente não há limitação no espaço físico para alocar registradores dentro da CPU, por que não colocar 256 registradores na CPU e ter muito mais dados para trabalhar?

2º Princípio de Projeto:
Menor é Mais Rápido!

  - Muitos registradores iriam aumentar o período de clock simplesmente porque os sinais elétricos teriam que percorrer um caminho maior
  - Também causariam um aumento no formado das instruções

Operandos: registradores!

- Convenção de nomes de registradores em RISC-V:

RISC-V Calling Convention			
Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries

Operandos: registradores!

- Você entendeu os nomes e usos dos registradores? Então refaça a conta abaixo usando registradores!

f = (g + h) – (i + j);

  - Especifique que registrador será utilizado para cada variável (e explique o motivo) – use a tabela!
  - Reescreva com os registradores

RISC-V Calling Convention			
Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries

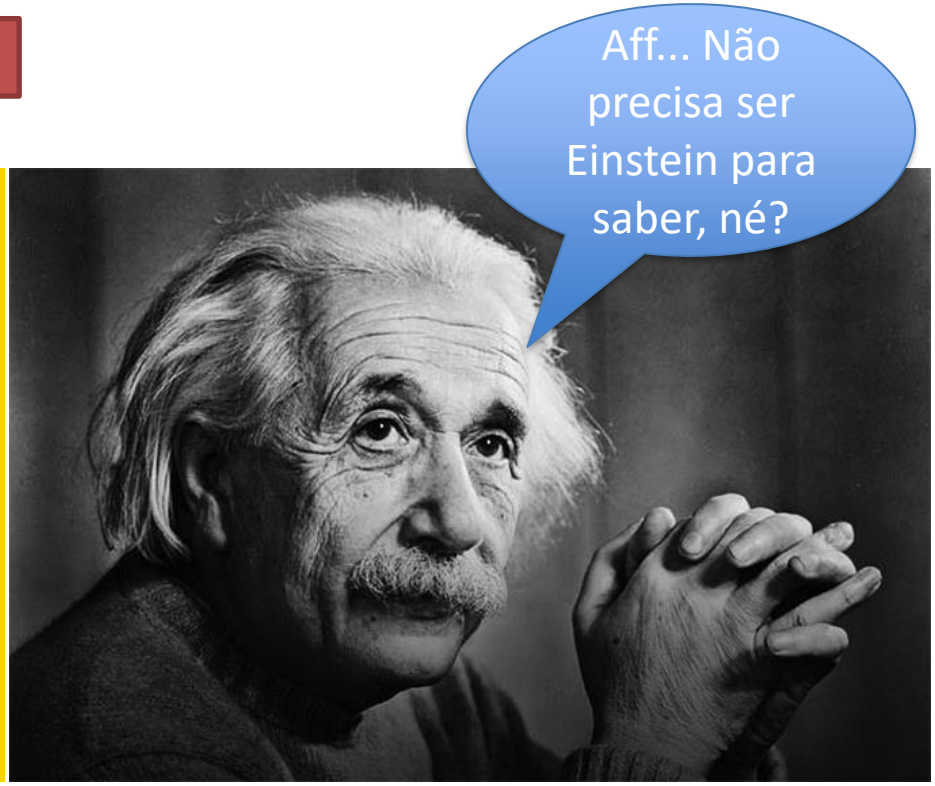
Operandos: memória!

- Quantos bits/bytes podem ficar armazenados nos registradores ao mesmo tempo?

Operandos: memória!

- Quantos bits/bytes podem ficar armazenados nos registradores ao mesmo tempo?

32 registradores x 32 bits = 1024 bits = 128 B



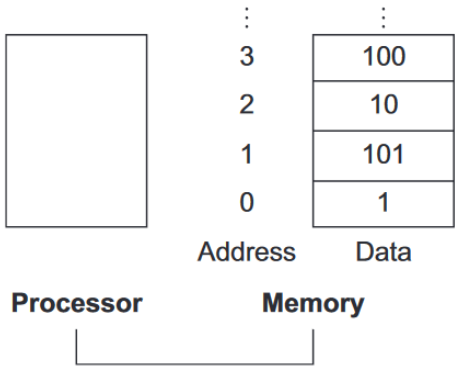
Operandos: memória!

- Estruturas de dados complexas (arrays, structs, estruturas dinâmicas, etc.) são mantidos na memória principal e transferidos para os registradores quando necessários através de uma **instrução de transferência de dados**.
  - Instrução de transferência de dados: uma instrução que move dados entre registradores e memória.
  - Instrução para buscar dado na memória e colocar em um registrador é a **`lw`** (de load word). ATENÇÃO: é complicado!

```
lw <reg.>, <offset>(<reg. endereço base>)
```

Operandos: memória!

- Para entender, vamos primeiro ver um exemplo SIMPLES, mas TOTALMENTE ERRADO. Depois vamos corrigir.
  - Imagine que você declarou o array A = [1, 101, 10, 100], com os seguintes endereços:



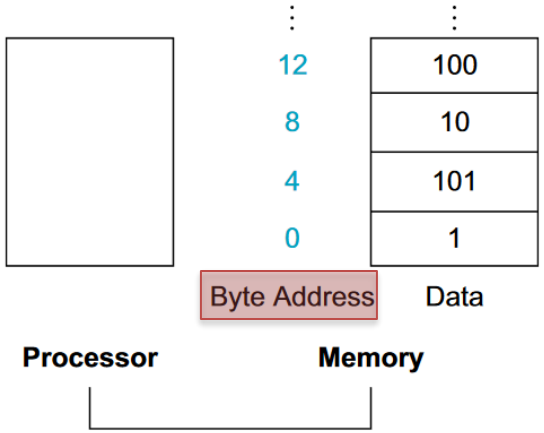
  - Como carregar o elemento A[2] (o valor 10) no registrador `x9`, considerando que o endereço base do array está no registrador `x22`? (o compilador/interpretador faz isso para nós)

`lw <reg.>, <offset>(<reg. endereço base>)`

`lw x9, 2(x22)`

Operandos: memória!

- Corrigindo agora:
  - Como uma word são 32 bits (4 bytes), os endereços dos elementos não são sequenciais. Array A = [1, 101, 10, 100]:



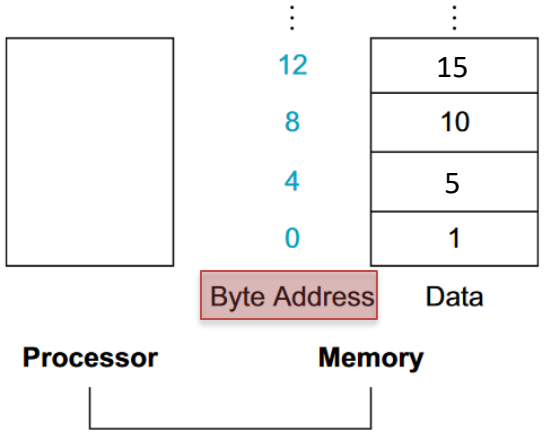
  - Como carregar o elemento A[2] (o valor 10) no registrador `x9`, considerando que o endereço base do array está no registrador `x22`? – Lembre-se: **o computador lê a word (4 bytes)**!

```
lw <reg.>, <offset>(<reg. endereço base>)

lw x9, 8(x22)
```

Operandos: memória!

- Sua vez:
  - Você declarou um Array A = [1, 5, 10, 15, 20, ..., 90, 95, 100], que recebeu os seguintes endereços de memória:

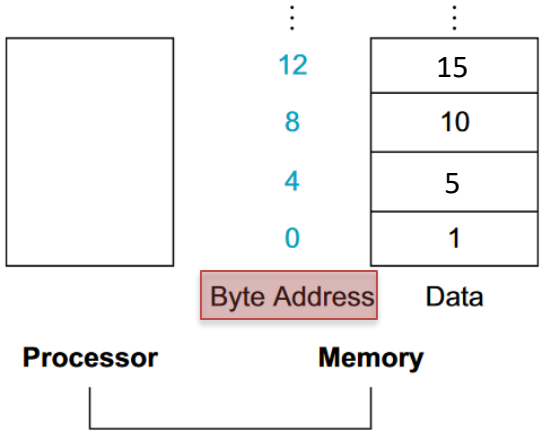


  - Como carregar o elemento A[13] no registrador `x5`, considerando que o endereço base do array está no registrador `x18`? – Lembre-se: **o computador lê a word (4 bytes)**! Qual o valor carregado em `x5`?

```
lw <reg.>, <offset>(<reg. endereço base>)
```

Operandos: memória!

- Sua vez:
  - Você declarou um Array A = [1, 5, 10, 15, 20, ..., 90, 95, 100], que recebeu os seguintes endereços de memória:



  - Como carregar o elemento A[13] no registrador `x5`, considerando que o endereço base do array está no registrador `x18`? – Lembre-se: **o computador lê a word (4 bytes)**! Qual o valor carregado em `x5`?

```
lw x5, 52(x18)
```

O valor carregado será o 65.

Operandos: memória!

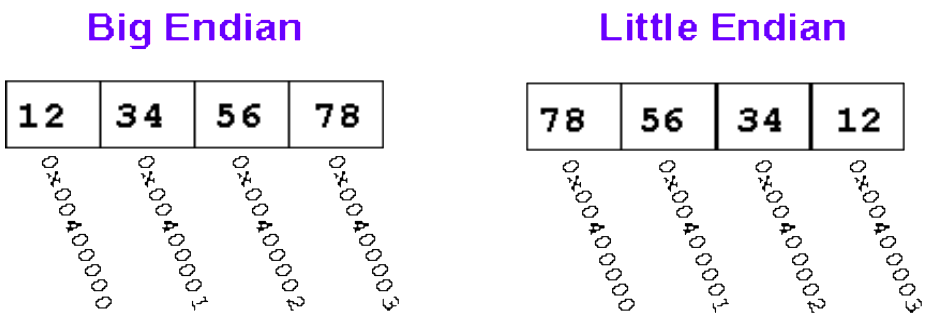
- Você entendeu mesmo? Você declarou, em C, um array do tipo double:

```
double A[N];  // elementos do tipo double
              // ocupam 8 bytes!
```

  - Como carregar o elemento A[23] no registrador `x6`, considerando que o endereço base do array está no registrador `x19`?

Operandos: memória!

- OK, vamos complicar um pouco agora! Como números grandes (que não cabem em uma word) são armazenados? Por exemplo, como armazenar o número 12345678?
- Existem 2 grandes categorias de computadores:
  - "Little End":
    - O byte MENOS significativo fica no MENOR endereço
  - "Big End"
    - O byte MAIS significativo fica no MENOR endereço
- RISC-V é little-endian!



Operandos: memória!

- Se existe uma operação para carregar um dado da memória para um registrar, deve existir uma operação inversa, ou seja, armazenar um dado do registrador na memória.
  - A instrução **`sw`** (store word) armazena um dado do registrador em uma posição na memória principal!
  - Funciona como o **`lw`**, mas agora a leitura é "invertida":

    `lw <reg.>, <offset>(<reg. endereço base>)`

  - Por exemplo: para salvar um número inteiro que está no registrador `x8` na posição A[10] de um array cujo endereço base está no registrador `x23`, fazemos:

    `sw x8, 40(x23)`

Operandos: memória!

- Juntando tudo: você declarou o seguinte array em C:

```
double A[N];  // elementos do tipo double
              // ocupam 8 bytes!
```

  - Como executar a seguinte operação:

A[12] = A[21] + A[33]

Obs.: considere que o endereço base do array está no registrador `x26`. Demonstre todas as operações (incluindo carregar com `lw`, as somas, e armazenar com `sw`).

Operandos: memória!

- Algumas observações:
  - O endereço base deve estar em um registrador
  - O offset deve ser um número constante
  - Atenção ao trabalhar com word ou doubleword, o offset muda (4 ou 8 bytes)
  - As instruções load word (`lw`) e store word (`sw`) são as duas únicas instruções que acessam a memória!
    - `lw` carrega uma word da memória para um registrador
    - `sw` armazena uma word de um registrador na memória
  - Compiladores tentam usar o máximo possível de registradores. Só "entornam" (spill) dados para a memória quando são pouco usados.

Operandos: constantes!

- Muitas operações aritméticas utilizam valores constantes, por exemplo:
  - Aumentar um contador
  - Controlar um loop
- RISC-V oferece diversas instruções com uma variação que permite trabalhar diretamente com operandos constantes.
  - Exemplo: f = f + 5

```
addi x22, x22, 5
```

  - Não existe uma instrução **`subi`**. Por quê?
  - Por que operandos constantes são importantes?

Operandos: constantes!

- Por que operandos constantes são importantes?

3º Princípio de Projeto:
Tornar Rápido o Caso Comum

  - Operações com constantes são extremamente comuns nos programas
  - Evita uma instrução de carregamento

Operandos: a constante 0 (zero)

- O registrador **x0** é sempre o valor 0 (zero). Não pode ser modificado! Muito útil para algumas operações, como:
  - Negar um número:

```
sub x9, x0, x21
```

  - Mover dados entre registradores:

```
add x9, x5, x0

addi x9, x5, 0
```

Resumo Até Aqui:

- Revisão sobre ISA
- Diferentes tipos de ISA
- Instruções para operações aritméticas básicas:
  - `add`
  - `addi`
  - `sub`
- Três Operandos:
  - Registradores: `x21`, `x5`, etc.
  - Memória: `4(x18)`
  - Constantes
- Instruções para transferência da dados (com endereço base e offset):
  - `lw`
  - `sw`

Últimas observações:

- Trabalharemos com o RV32, que é uma versão de 32 bits do RISC-V (a palavra é de 32 bits). Também existe a versão RV64 do RISC-V (a palavra é de 64 bits).

- É importante verificar o tamanho do tipo de dados, principalmente em arquiteturas de 64 bits:

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

Hora de Esfriar a Cabeça!