DCC006 – Organização de Computadores I

Aula 3 – Instruções

A linguagem do Computador

Prof. Omar Paranaiba Vilela Neto

DCC

DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Relembrando

Linguagem de alto nível

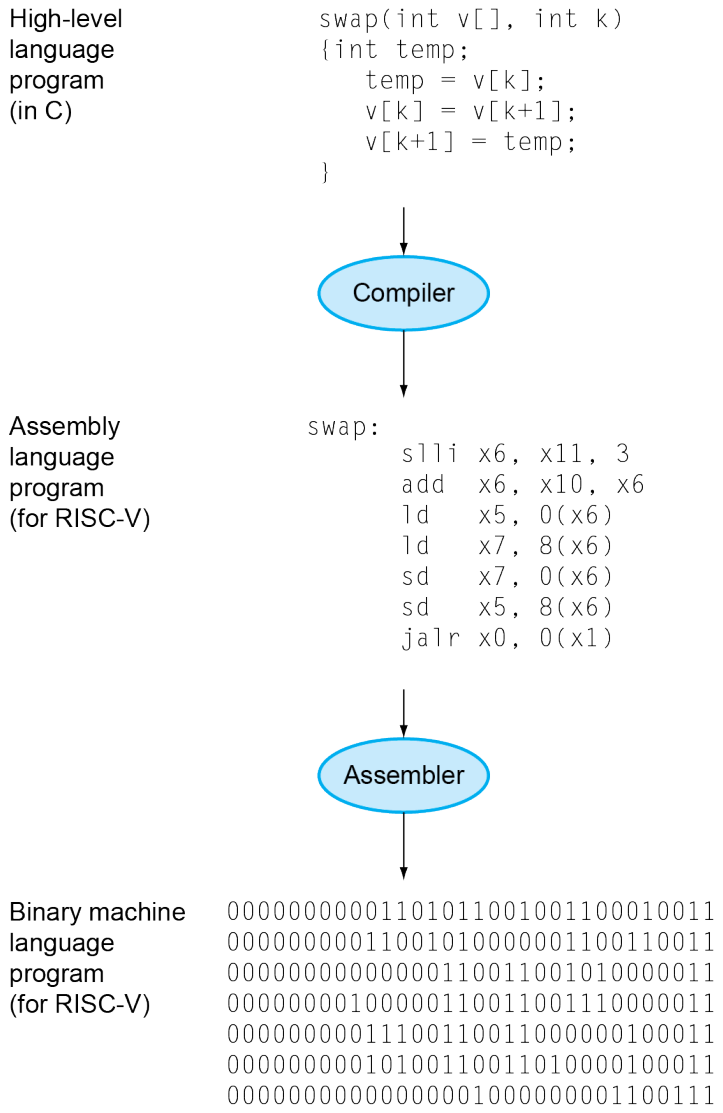
- Nível de abstração próximo do domínio do problema
- Focado em produtividade e portabilidade

Linguagem Assembly

- Representação textual de instruções

Linguagem de Hardware

- Dígitos binários (bits)
- Codifica instruções e dados



Conjunto de Instruções - ISA

- O repertório de instruções de um computador
- Computadores diferentes possuem conjuntos de instruções diferentes
  - Mas comum em muitos aspectos
- Computadores recentes tinham conjuntos de instruções muito simples
  - Implementações simplificadas
- Muitos computadores modernos também tem conjuntos de instruções muito simples

Conjunto de Instruções do RISC-V

- Usado como exemplo neste curso
- Desenvolvido na UC Berkeley como ISA aberto
  - 2010
- Agora gerenciado pela RISC-V Foundation (riscv.org)
- Típico de muitos ISAs modernos
  - Veja o cartão de referência do RISC-V (Moodle)
- ISAs similares possuem um vasto mercado em sistemas embarcados
  - Aplicações em eletrônicos, equipamento de rede/ armazenamento, câmeras, impressoras, …

RISC-V - Aritmética

- **Todas instruções tem 3 operandos**
- **A ordem dos operandos é fixa (destino primeiro)**

Exemplo:

Código C: `A = B + C`

Código RISC-V: `add A, B, C`

- **Operandos devem ser registradores, só existem 32 registradores**
  - **Tamanho dos registradores: 64 bits. (Double world)**

- **Principio de Projeto 1: Simplicidade favorece regularidade**
  - **Razão para poucos registradores**

RISC-V - Registradores

- **Operandos de instruções Aritméticas devem ser registradores,**
  - **— só 32 registradores existem**
- **Compilador associa variáveis com registradores**

Registradores

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

RISC-V - Aritmética

- **Todas instruções tem 3 operandos**
- **A ordem dos operandos é fixa (destino primeiro)**

Exemplo:

Código C: `A = B + C`

Código RISC-V: `add x5, x6, x7`

(associação com variáveis pelo compilador)

- **Operandos devem ser registradores, só existem 32 registradores**
  - **Tamanho dos registradores: 32 bit. (MIPS-32)**

- **Principio de Projeto 2: Menor é mais rápido.**
  - **Razão para poucos registradores**

RISC-V - Aritmética

- **Principio de projeto:** **simplicidade favorece a regularidade.**

- **Código C:**

```
F = (G + H) - (I - J)
```

  - F,...,J em x19,...,x23

- **Código Intermediário:**

```
add t0, G, H
add t1, I, J
sub F, t0, t1
```

- **Código RISC-V:**

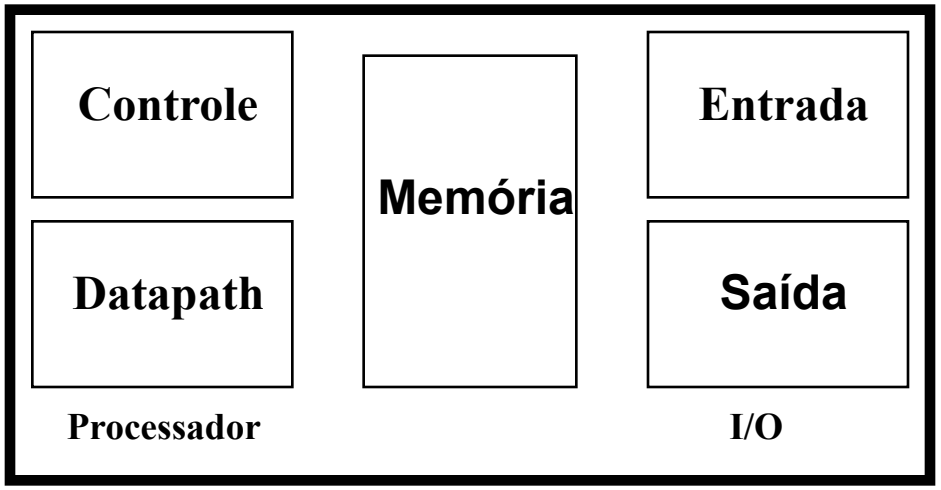
```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

RISC-V - Aritmética

Como fazer quando um programa tem muitas variáveis?

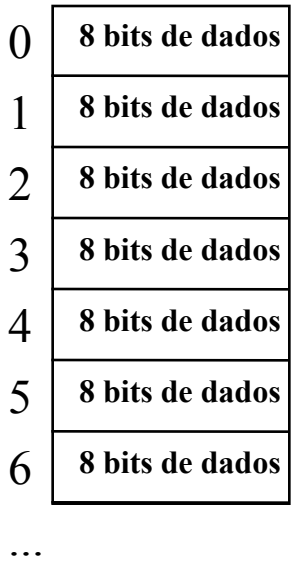
Registradores x Memória

Por quê não usamos a Memória?



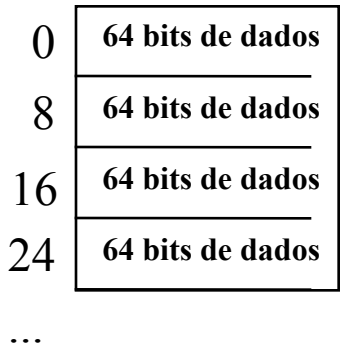
Organização da Memória

- **Visto como uma matriz unidimensional, com um endereço.**
- **Um endereço de memória é um índice em uma matriz**
- **endereçamento de Byte aponta para um byte da memória.**



Organização da Memória

- **Bytes são pequenos, mas vários dados usam ” double words”**
- **Para o RISC-V, uma palavra dupla tem 64 bits ou 8 bytes.**



Registradores retém 64 bits de dados

- **2^{32} bytes com endereço de byte de 0 to 2^{32}-1**
- **2^{29} palavras duplas com endereço de byte 0, 8, 16, ... 2^{32}-8**

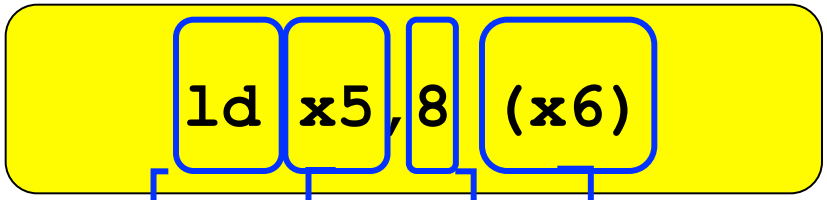
RISC-V – Acesso à Memória

- **Instruções load e store**
- **Load – Carrega conteúdo da memória para o registrador**
  - **ld x5, 8(x6)**
- **Store – Copia conteúdo do registrador na memória**
  - **sd x5, 8(x6)**

RISC-V – Acesso à Memória

Copiar dados de → para	Instrução
Memória → Registrador	load word (ld)
Registrador → Memória	store word (sd)

Formato:



instrução

registrador de destino

offset (em bytes)

registrador base, que armazena o endereço base

RISC-V – Acesso à Memória

- **Instruções load e store**
- **Exemplo:**

Código C: `A[12] = h + A[8];`

Código MIPS:

```
ld   x9, 64(x22)
add  x9, x21, x9
sd   x9, 96(x22)
```

- **Store tem destino por último**
- **Relembre operandos aritméticos são registradores, não memória!**

RISC-V – Acesso à Memória

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

RISC-V – Aritmética com imediatos

- **Todas instruções tem 3 operandos**
- **Um operando pode ser um número imediato**

- **Exemplo:**

  Código C: `A = B + 21`

  Código RISC-V: `add x5, x6, 21`

  (associação com variáveis pelo compilador)

- **Faça o caso comum rápido**

  - **Pequenas constantes são comuns**
  - **Operando imediato evita um load**

Revisão: Números Inteiros

- **Bits são bits**
  - **— convenções define relação entre bits e números**
- **Números Binários (base 2)**

  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...

  decimal: $0...2^n-1$

- **mais complicado:**

  números são finitos (overflow)

  frações e números reais

  números negativos

  i.e., RISC-Vnão tem instrução subi; (addi pode somar um número negativo)

- **Como nós representamos um número negativo?**
  - **i.e., qual padrão representará os números?**

Revisão: Números Inteiros

- **Saídas: balanço, números de zeros, facilidade das operações**
- **Qual é a melhor? Porque?**

- **Sinal Magnitude:**
- **000 = +0**
- **001 = +1**
- **010 = +2**
- **011 = +3**
- **100 = -0**
- **101 = -1**
- **110 = -2**

111 = -3

- **Complemento de 1:**
- **000 = +0**
- **001 = +1**
- **010 = +2**
- **011 = +3**
- **100 = -3**
- **101 = -2**
- **110 = -1**

111 = -0

- **Complemento de 2:**
- **000 = +0**
- **001 = +1**
- **010 = +2**
- **011 = +3**
- **100 = -4**
- **101 = -3**
- **110 = -2**

111 = -1

Revisão: Números Inteiros

•32 bit complemento de dois:

•$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
•$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
•$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
•...
•$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
•$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
•$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
•$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
•$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
•...
•$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
•$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
•$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

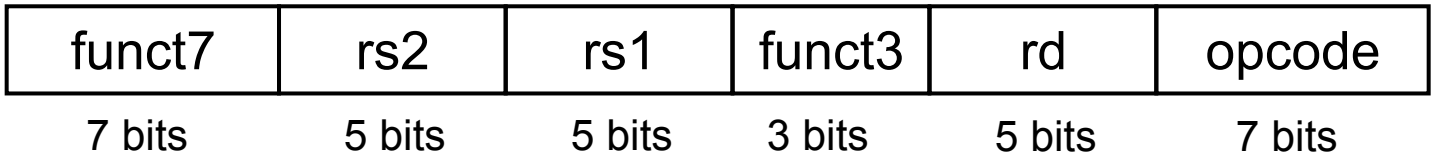
- No conjunto de instruções do RISC-V
  - lb: load byte com sinal
  - lbu: load byte sem sinal (positivo)

Traduzindo para a Linguagem da Máquina

- Instruções são codificadas em binário
  - Chamado de Código de Máquina
- Instruções RISC-V
  - Codificado como palavras de 32-bit
  - Pequeno número de formatos codificando: código da operação, número dos registradores, ...
  - Regularidade!

Traduzindo para a Linguagem da Máquina

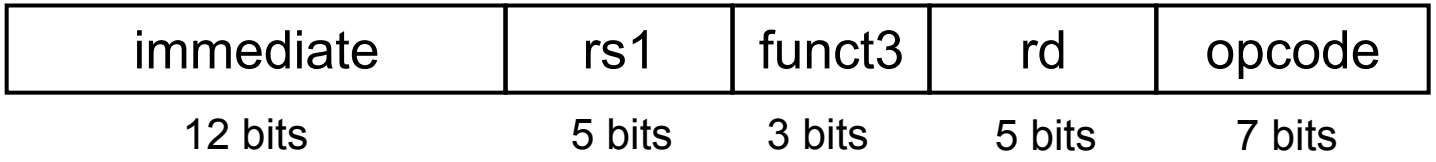
Instruções Tipo-R



- Instruction fields
  - opcode: código da operação
  - rd: número do registrador de destino
  - funct3: 3-bit código de função (adicional ao opcode)
  - rs1: número do primeiro registrador de origem
  - rs2: número do segundoregistrador de origem
  - funct7: 7-bit código de função (adicional ao opcode)

Traduzindo para a Linguagem da Máquina

Instruções Tipo-I

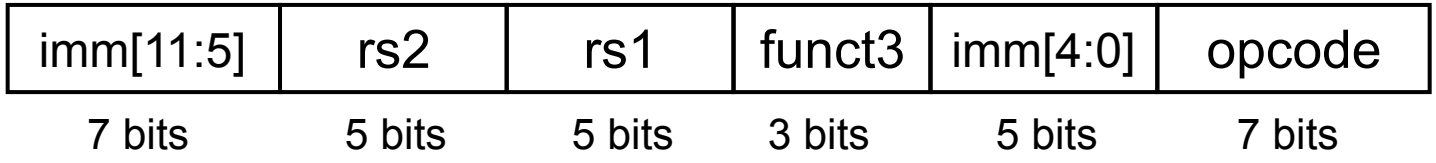


- Aritmética imediata e load
  - rs1: número do registrador de origem ou base
  - immediate: operando constante ou offset adicionado à base
    - Complemento de 2, valor será estendido

- **Principio de Projeto 3: Bons projetos demandam bons compromissos.**
  - **Diferentes formatos complica a decodificação, mas permite instruções uniformes de 32 bits**
  - **Manter os formatos o mais similar possível.**

Traduzindo para a Linguagem da Máquina

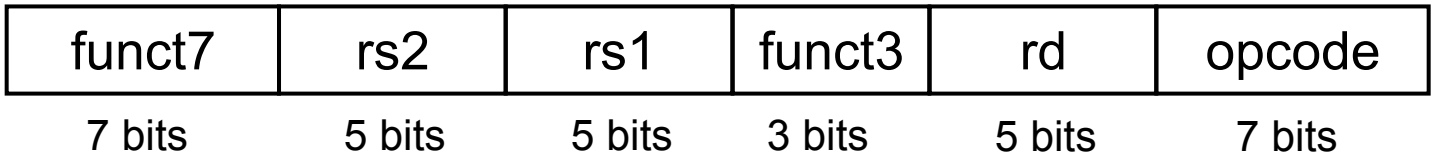
Instruções Tipo-S



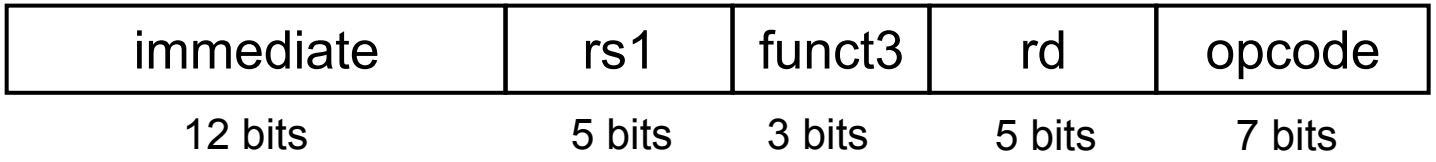
- Different immediate format for store instructions
  - rs1: número do registrador de base
  - rs2: número do registrador de origem
  - immediate: offset adicionado à base
    - Separado para que rs1 e rs2 estejam sempre no mesmo local

Traduzindo para a Linguagem da Máquina

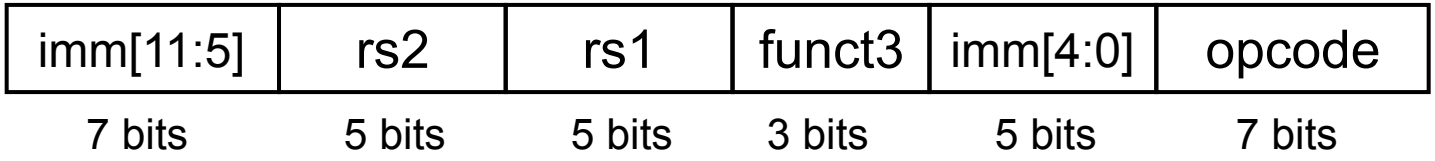
Instruções Tipo-R



Instruções Tipo-I



Instruções Tipo-S



Traduzindo para a Linguagem da Máquina

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (Add)	R	0000000	reg	reg	000	reg	0110011
sub (Sub)	R	0100000	reg	reg		reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (Add Immediate)	I	constant		reg	000	reg	0010011
ld (Load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sd (Store doubleword)	S	address	reg	reg	011	address	0100011

Traduzindo para a Linguagem da Máquina

Exemplo Tipo-R

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
add x9,x20,x21
```

0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

Traduzindo para a Linguagem da Máquina

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

Traduzindo para a Linguagem da Máquina

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

Assembler

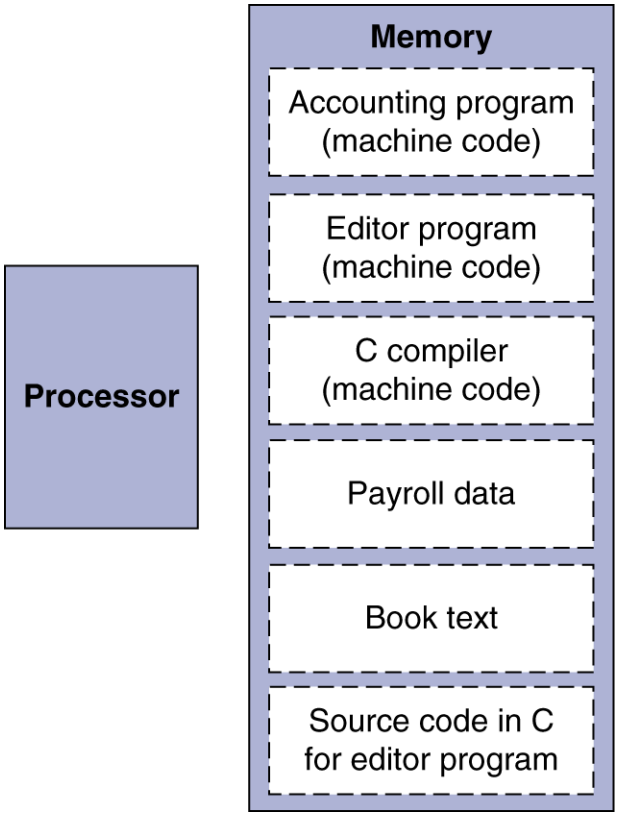
Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000011001100101000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

Agora vocês entendem!

Armazenamento de Programas

The BIG Picture



- Instruções representadas em binários, como dados.
- Instruções e dados armazenados na memória
- Programas podem operar em programas
  - ex., Compiladores, linkadores, …
- Compatibilidade binária permite programas compilados funcionarem em computadores diferentes
  - ISAs Padronizados

RISC-V – Operações Lógicas

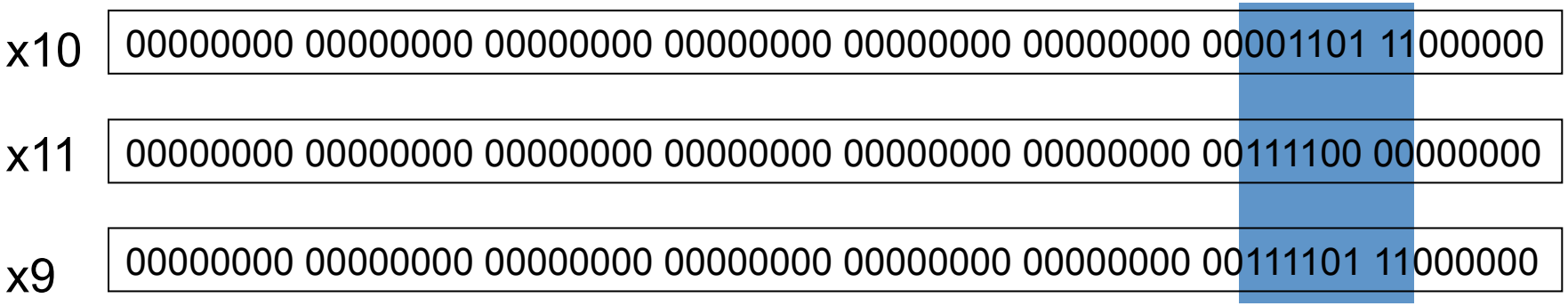
Instruções para manipulação de bits

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

RISC-V – Operações Lógicas

Exemplo OR

```
or x9,x10,x11
```



RISC-V – Operações Condicionais

- Desvia para a instrução marcada se a condição for verdadeira
  - Caso contrário, continua sequencialmente

- `beq rs1, rs2, L1`
  - se (rs1 == rs2) desvia para a instrução marcada L1

- `bne rs1, rs2, L1`
  - se (rs1 != rs2) desvia para a instrução marcada L1

- `blt rs1, rs2, L1`
  - Se (rs1 < rs2) desvia para a instrução marcada L1

- `bge rs1, rs2, L1`
  - se (rs1 >= rs2) desvia para a instrução marcada L1

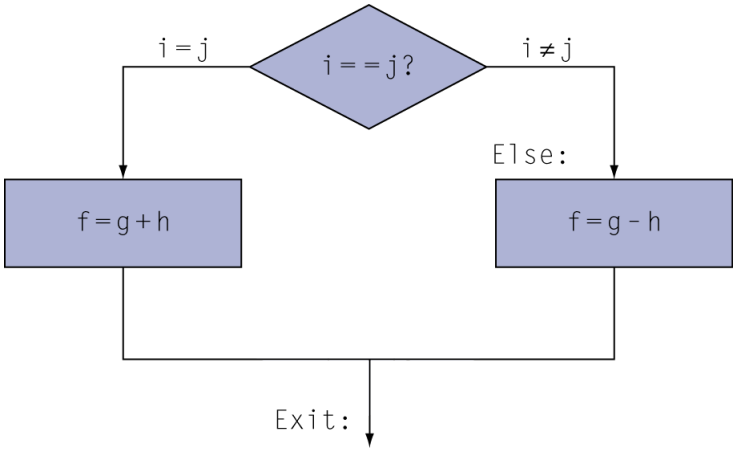
RISC-V – Operações Condicionais

Exemplo IF

- C:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … em x19, x20, …



- RISC-V:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // incondicional
Else:   sub x19, x20, x21
Exit:   …
```

RISC-V – Operações Condicionais

Exemplo Loop

- C:

```
while (save[i] == k) i += 1;
```

  - i em x22, k em x24, endereço de save em x25

- RISC-V:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

RISC-V – Chamada de Procedimento

- Procedimentos: Conjunto de instruções com função definida
- Realizam uma série de operações como base em valores de parâmetros
- Podem retornar valores computados
- Motivos para o uso de procedimentos:
  - Tornar o programa mais fácil de ser entendido
  - Permitir a reutilização do código do procedimento
  - Permitir que o programador se concentre em uma parte do código (os parâmetros funcionam como barreira)

RISC-V – Chamada de Procedimento

Etapas para execução de um procedimento

1 - Colocar parâmetros em um lugar onde o procedimento pode acessar;

2 - Transferir o controle para o procedimento

3 - Adquirir os recursos de armazenamento necessários para o procedimento

4 - Realizar a tarefa desejada

5 - Colocar o valor de retorno em um local onde o programa que chamou o procedimento possa acessá-lo

6 - Retornar o controle para o ponto de origem. Um procedimento pode ser chamado por vários pontos em um programa.

RISC-V – Chamada de Procedimento

Registradores de procedimento

- x12-x17: registradores de argumento – passa os parâmetros;
- x10-x11: registradores de valor – valores de retorno;
- x1: registrador de endereço de retorno – retorna ao ponto de origem;
- PC: contador de programa – guarda o endereço da instrução executada.

Instruções de procedimento

- jal x1
  - Pula para o procedimento e coloca o endereço de retorno em x1
    - x1 = PC + 4
- jalr x0, 0(x1)
  - Volta para endereço armazenado em x1

RISC-V – Chamada de Procedimento

O que fazer se um procedimento precisar de mais registradores?

Algumas opções:

- x5-x7, x28-x31: registradores temporários não preservados;
- x9, x18-x27: registradores salvos que precisam ser preservados.

Onde preservar os registradores salvos?

Em uma pilha na memória

- sp: stack pointer – apontador de pilha.

RISC-V – Chamada de Procedimento

Exemplo

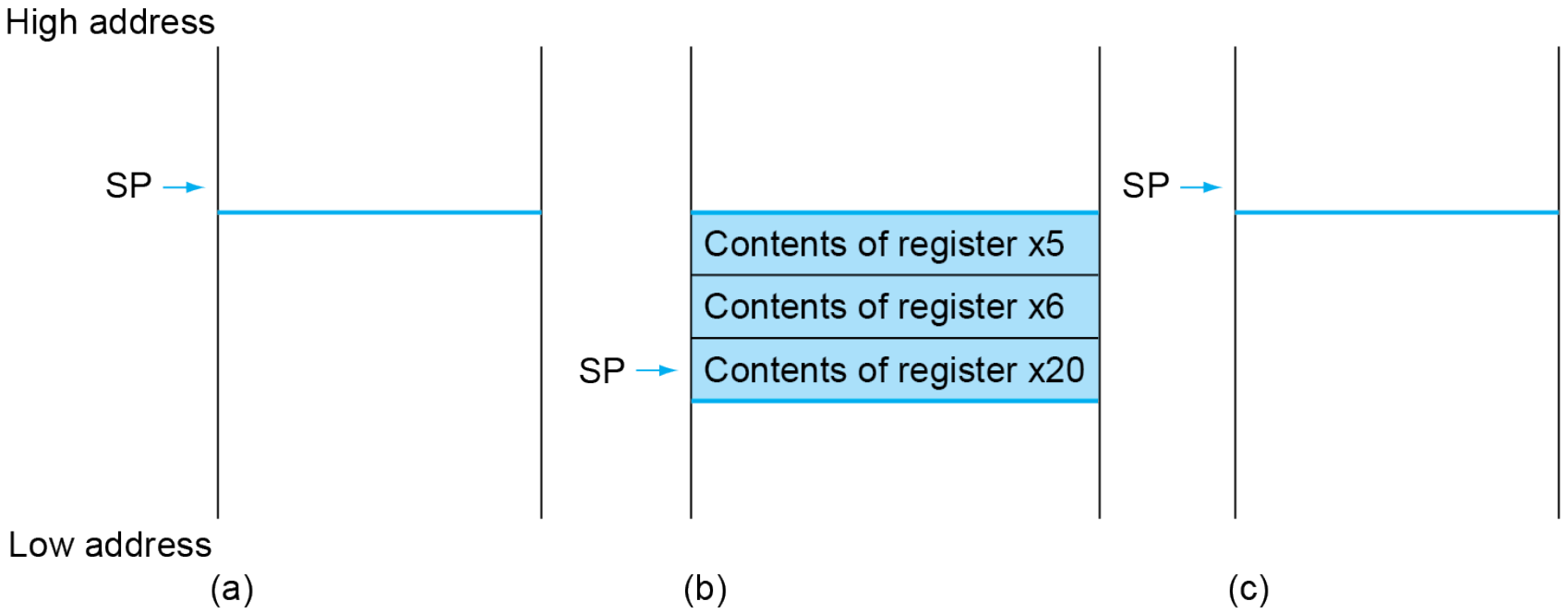
```
int folha (int g, int h, int i, int j)
{
        int f;

        f = (g+h) – (i+j);
        return f
}
```

RISC-V code:

```
folha:
        addi sp,sp,-24
        sd   x5,16(sp)
        sd   x6,8(sp)
        sd   x20,0(sp)
        add  x5,x10,x11
        add  x6,x12,x1
        sub  x20,x5,x6
        addi x10,x20,0
        ld   x20,0(sp)
        ld   x6,8(sp)
        ld   x5,16(sp)
        addi sp,sp,24
        jalr x0,0(x1)
```

RISC-V – Chamada de Procedimento



RISC-V – Chamada de Procedimento

Exemplo – Chama de procedimento – Recursivo

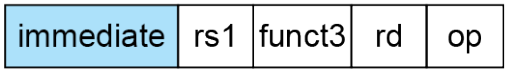
```
int fact (int n)
{
        if (n < 1) return (1);
        else return (n * fact(n-1));
}
```

RISC-V code:

```
fact:
        addi sp,sp,-16
        sd   x1,8(sp)
        sd   x10,0(sp)
        addi x5,x10,-1
        bge  x5,x0,L1
        addi x10,x0,1
        addi sp,sp,16
        jalr x0,0(x1)
L1: addi x10,x10,-1
        jal  x1,fact
        addi x6,x10,0
        ld   x10,0(sp)
        ld   x1,8(sp)
        addi sp,sp,16
        mul  x10,x10,x6
        jalr x0,0(x1)
```

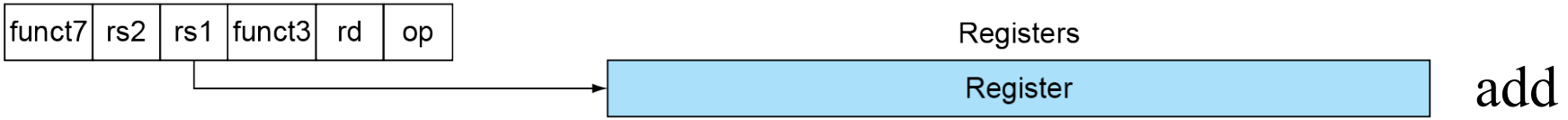
RISC-V – Endereçamento

1. Immediate addressing



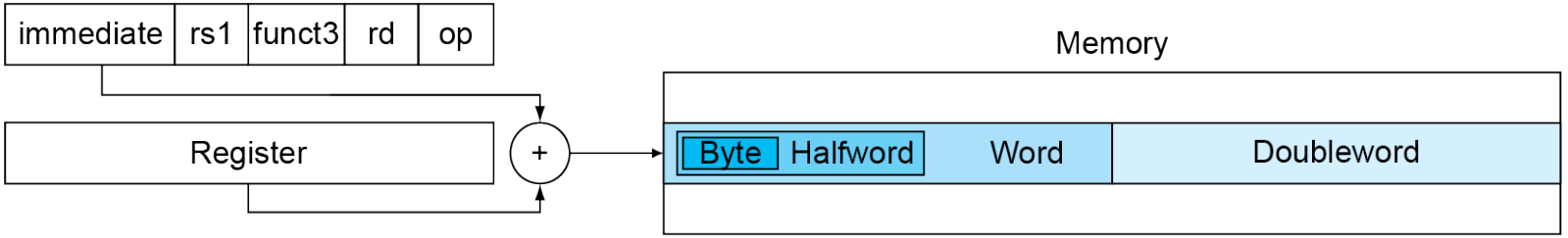
addi

2. Register addressing



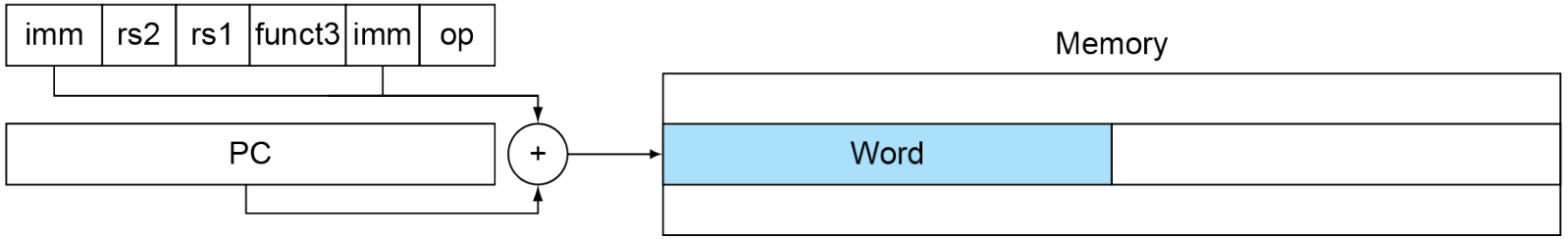
add

3. Base addressing



lw

4. PC-relative addressing



j bne

RISC-V – Array x Ponteiro

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```