Introduction to Assembly and RISC-V

Reminders:

- Lab 1 released today
- Lab hours begin today
- Sign up for piazza

“General Purpose” Processor

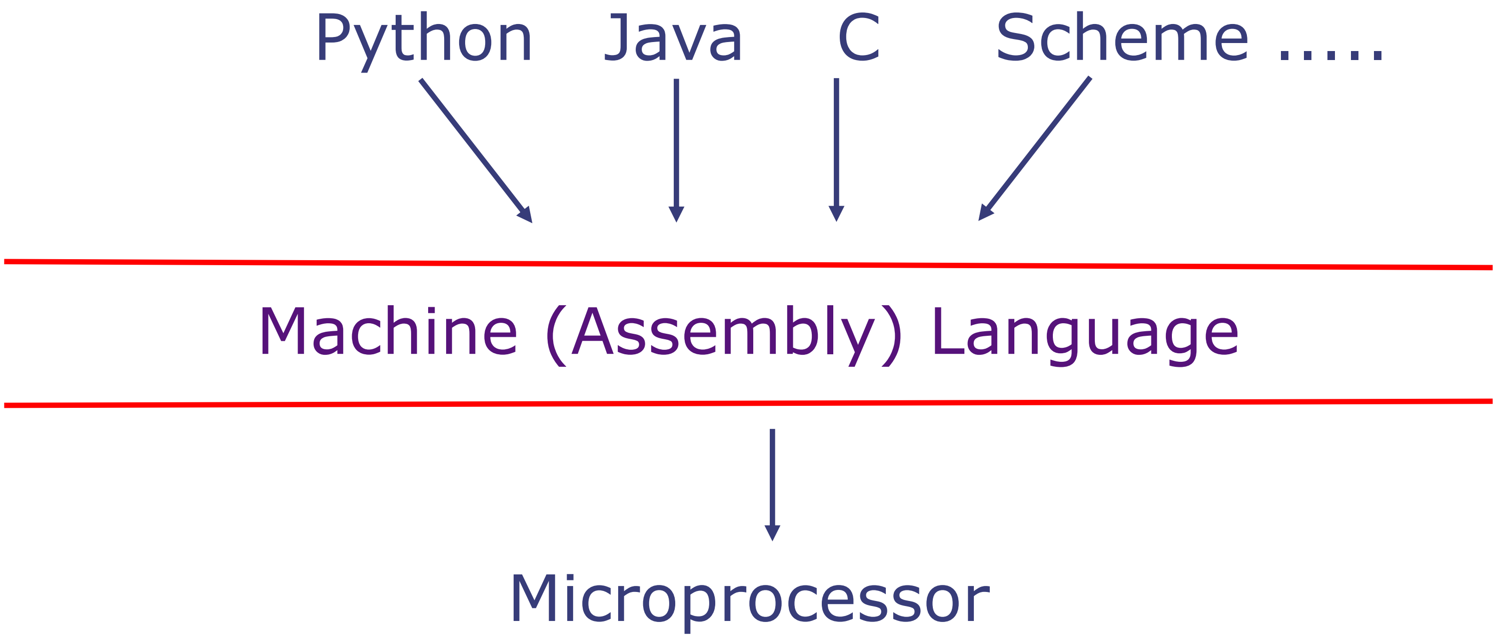
- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language

“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
- It is also not sensible to execute every feature of a high-level language directly in hardware

“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
- It is also not sensible to execute every feature of a high-level language directly in hardware



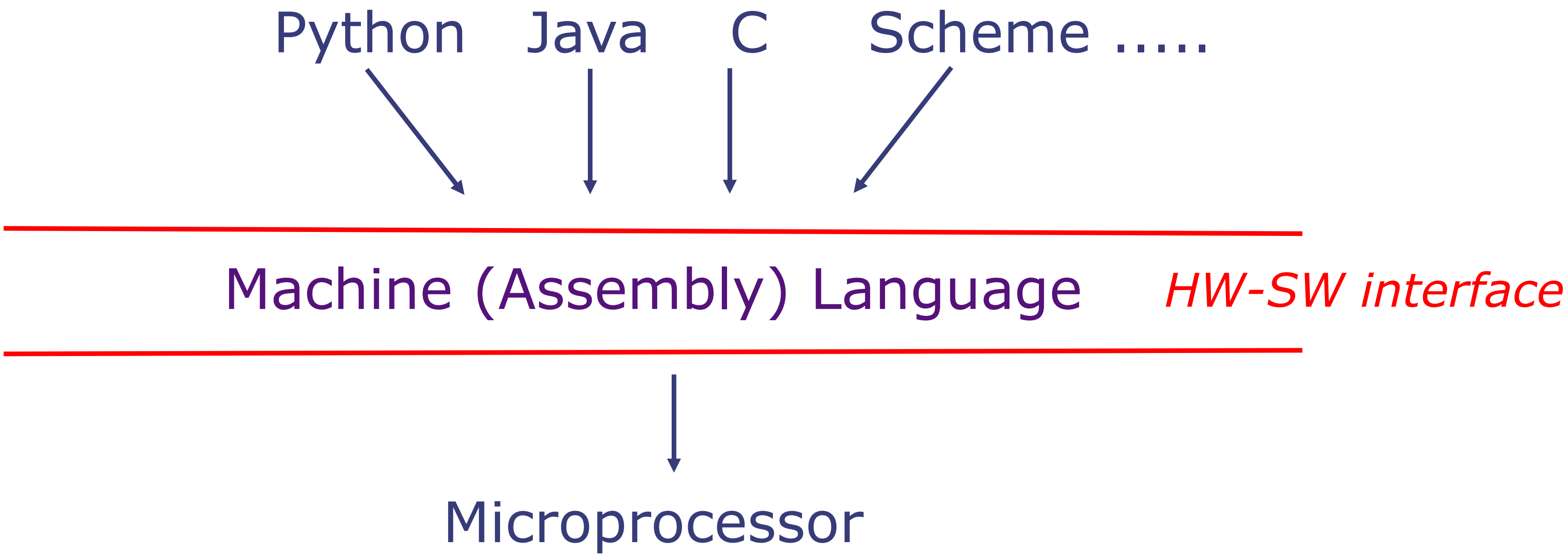
Python Java C Scheme .....

Machine (Assembly) Language

Microprocessor

"General Purpose" Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
- It is also not sensible to execute every feature of a high-level language directly in hardware



Python Java C Scheme .....

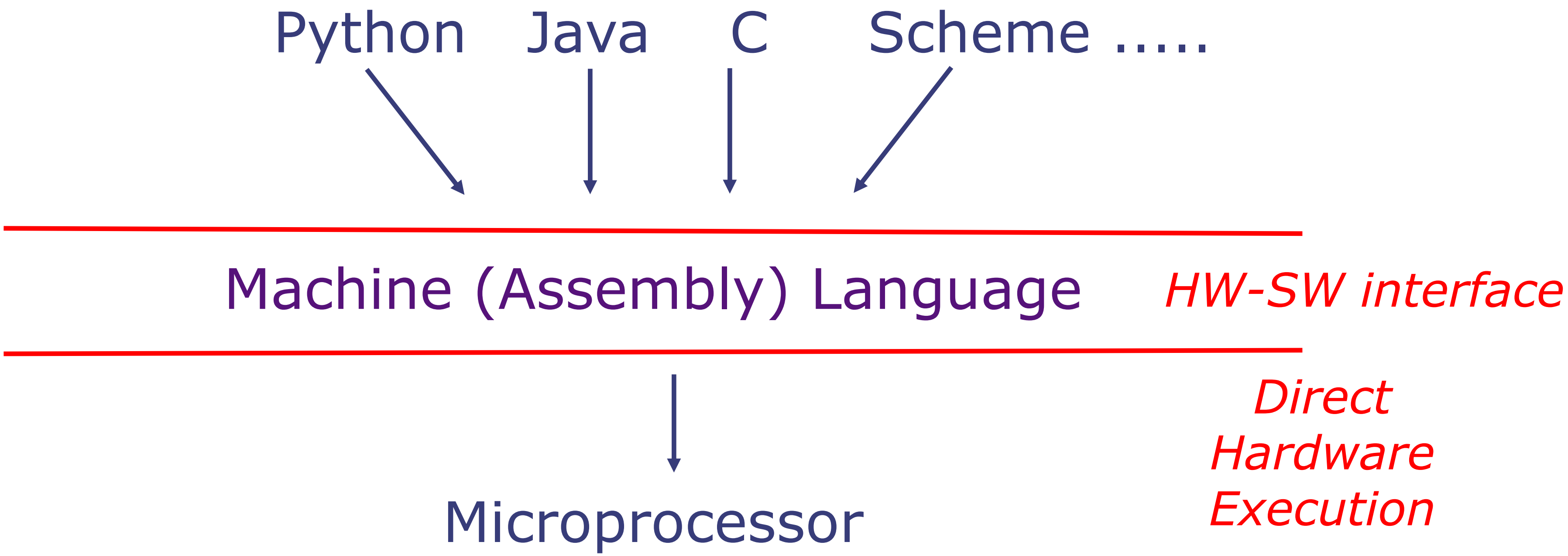
Machine (Assembly) Language

HW-SW interface

Microprocessor

“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
- It is also not sensible to execute every feature of a high-level language directly in hardware



Python Java C Scheme .....

Machine (Assembly) Language

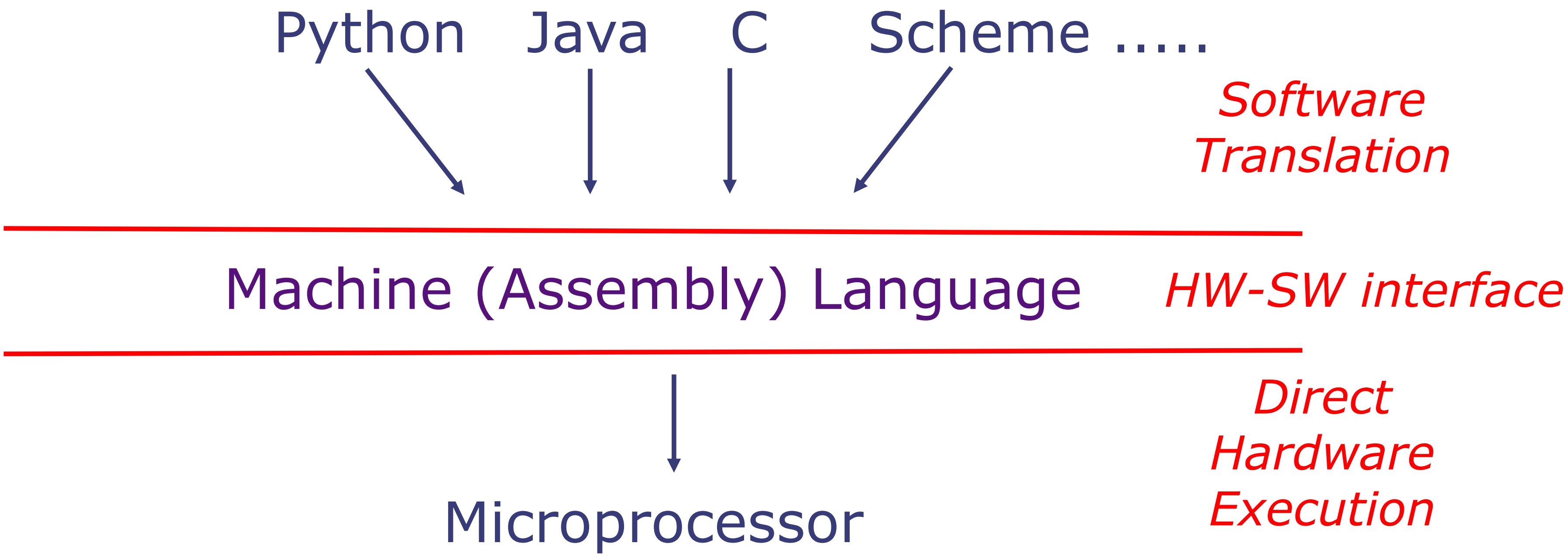
HW-SW interface

Microprocessor

Direct Hardware Execution

“General Purpose” Processor

- It would be highly desirable if the same hardware could execute programs written in Python, Java, C, or any high-level language
- It is also not sensible to execute every feature of a high-level language directly in hardware



Python Java C Scheme .....

Software Translation

Machine (Assembly) Language

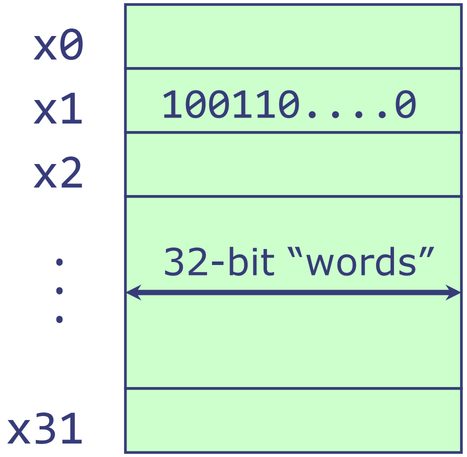
HW-SW interface

Direct Hardware Execution

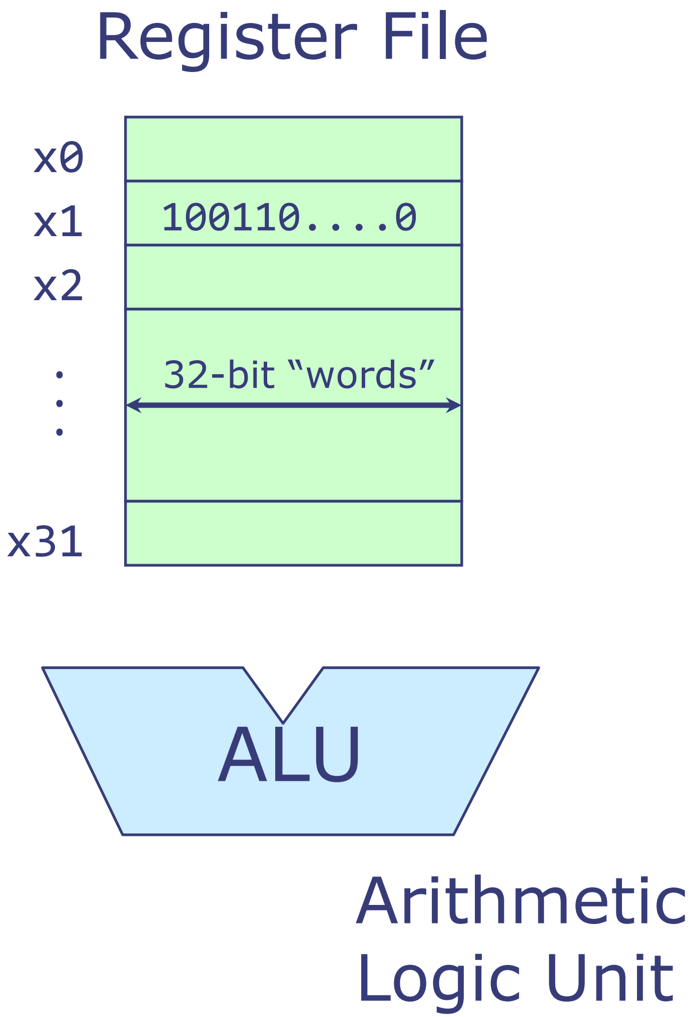
Microprocessor

Components of a MicroProcessor

Register File

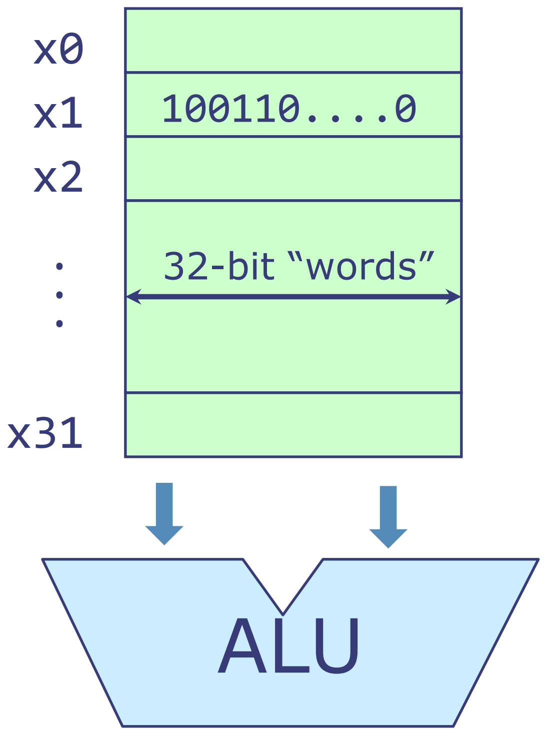


Components of a MicroProcessor



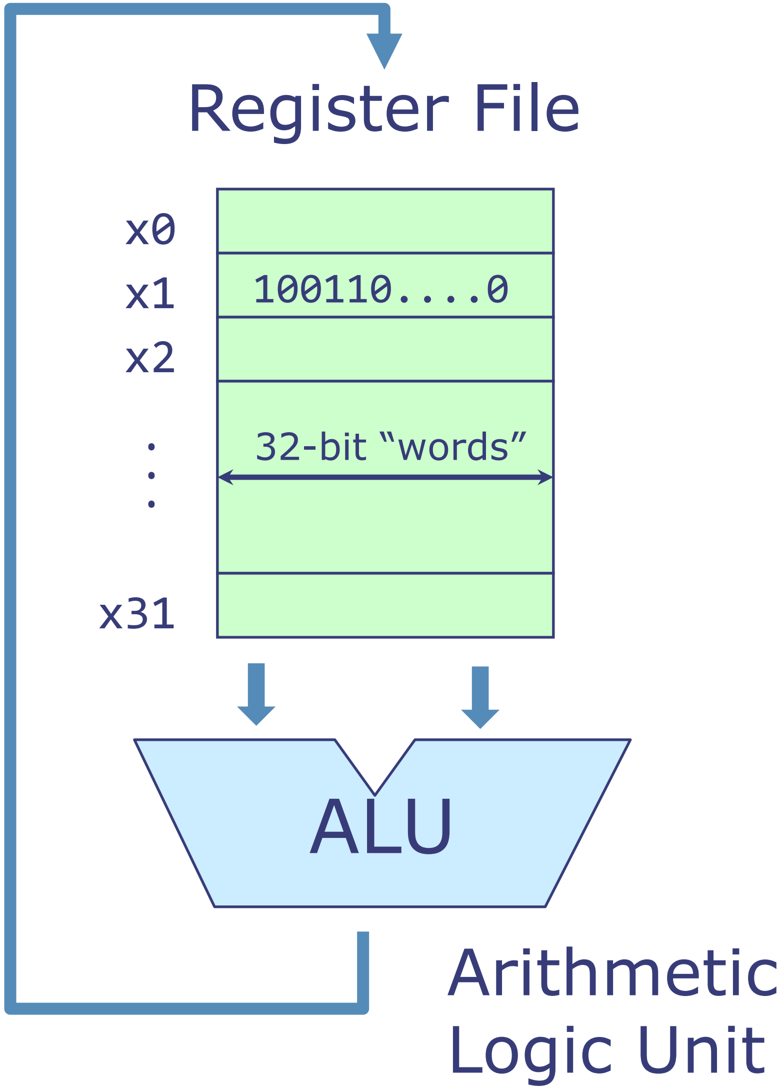
Components of a MicroProcessor

Register File



Arithmetic
Logic Unit

Components of a MicroProcessor



Register File

x0

x1 100110....0

x2

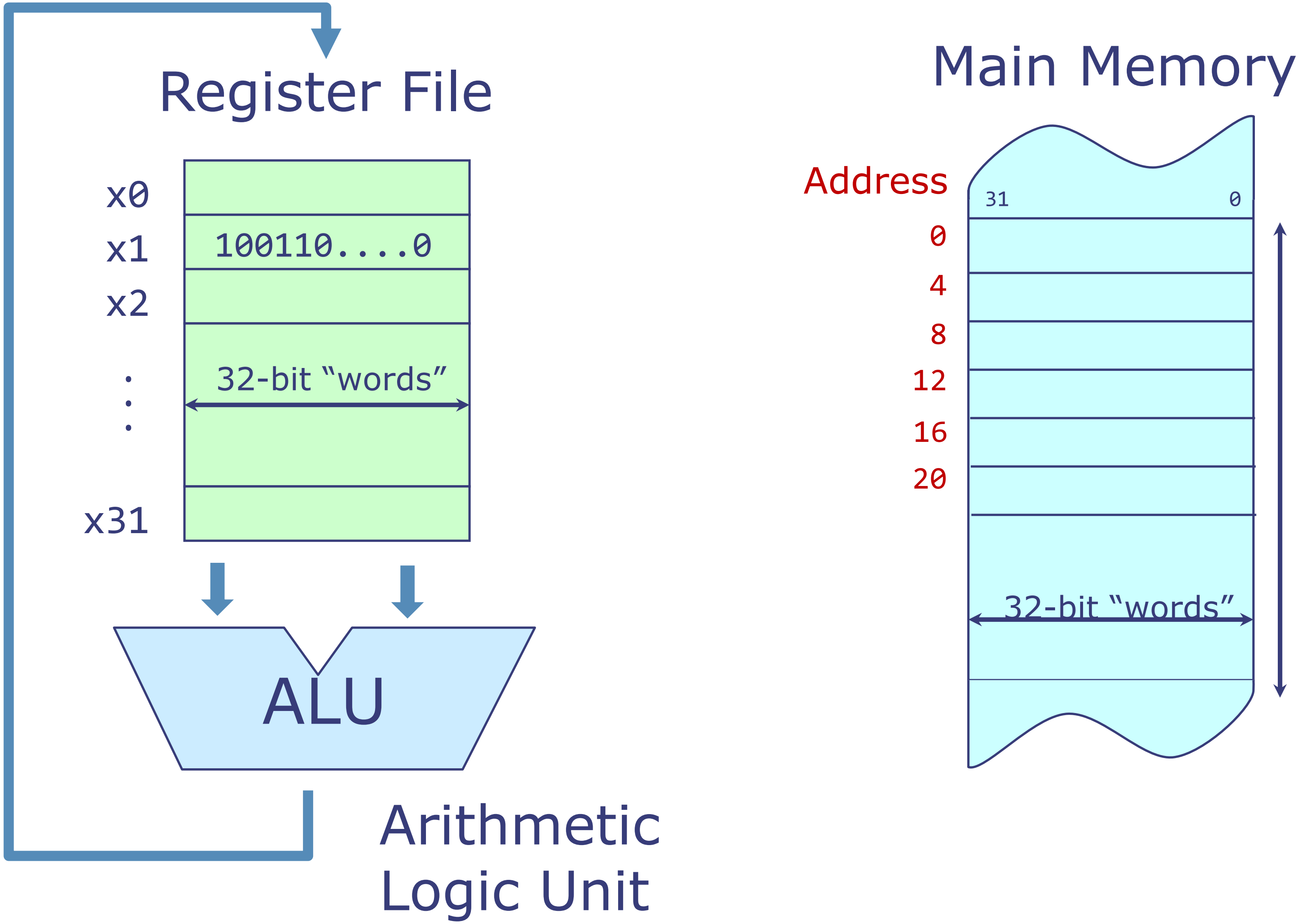
32-bit "words"

x31

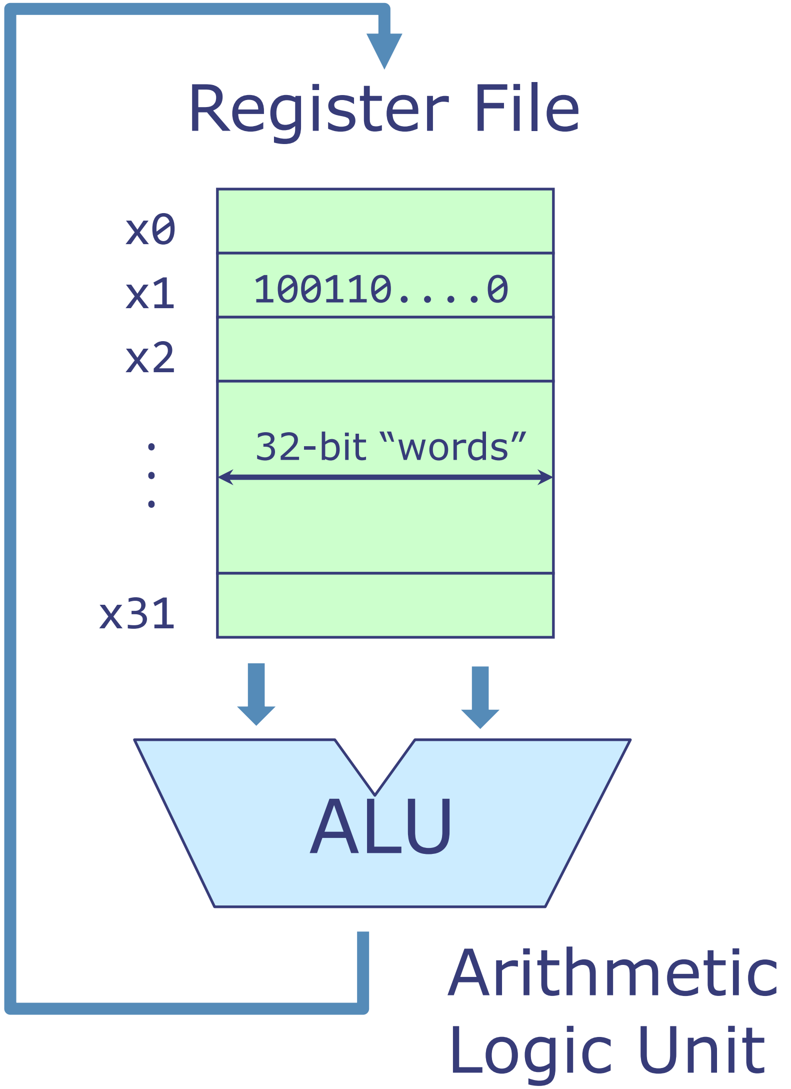
ALU

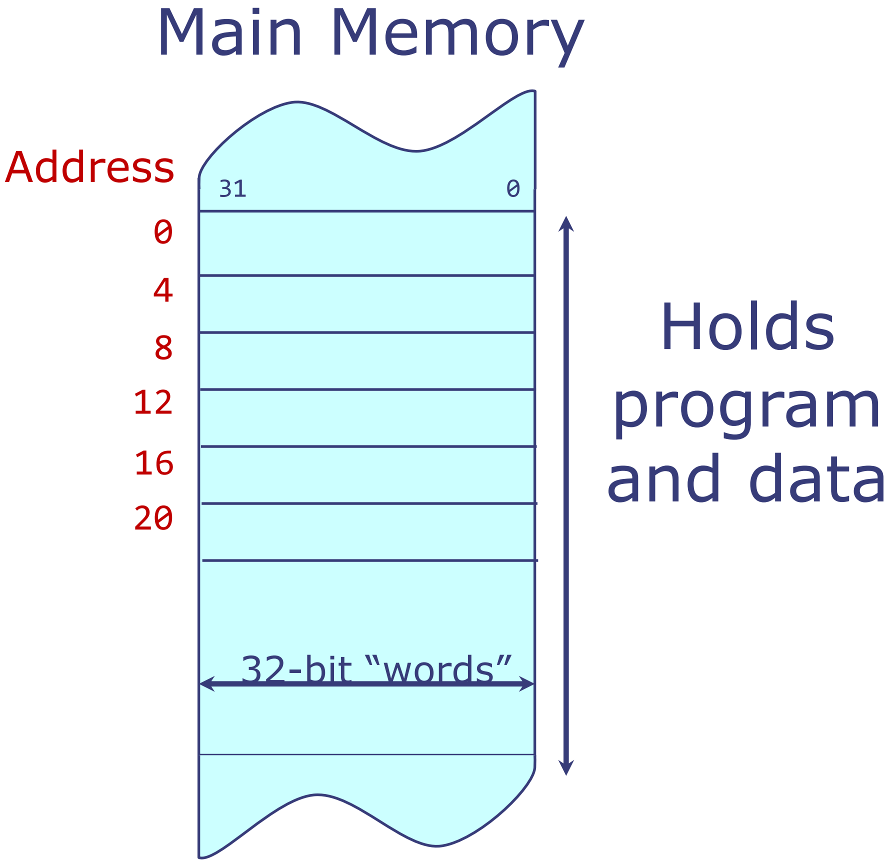
Arithmetic Logic Unit

Components of a MicroProcessor

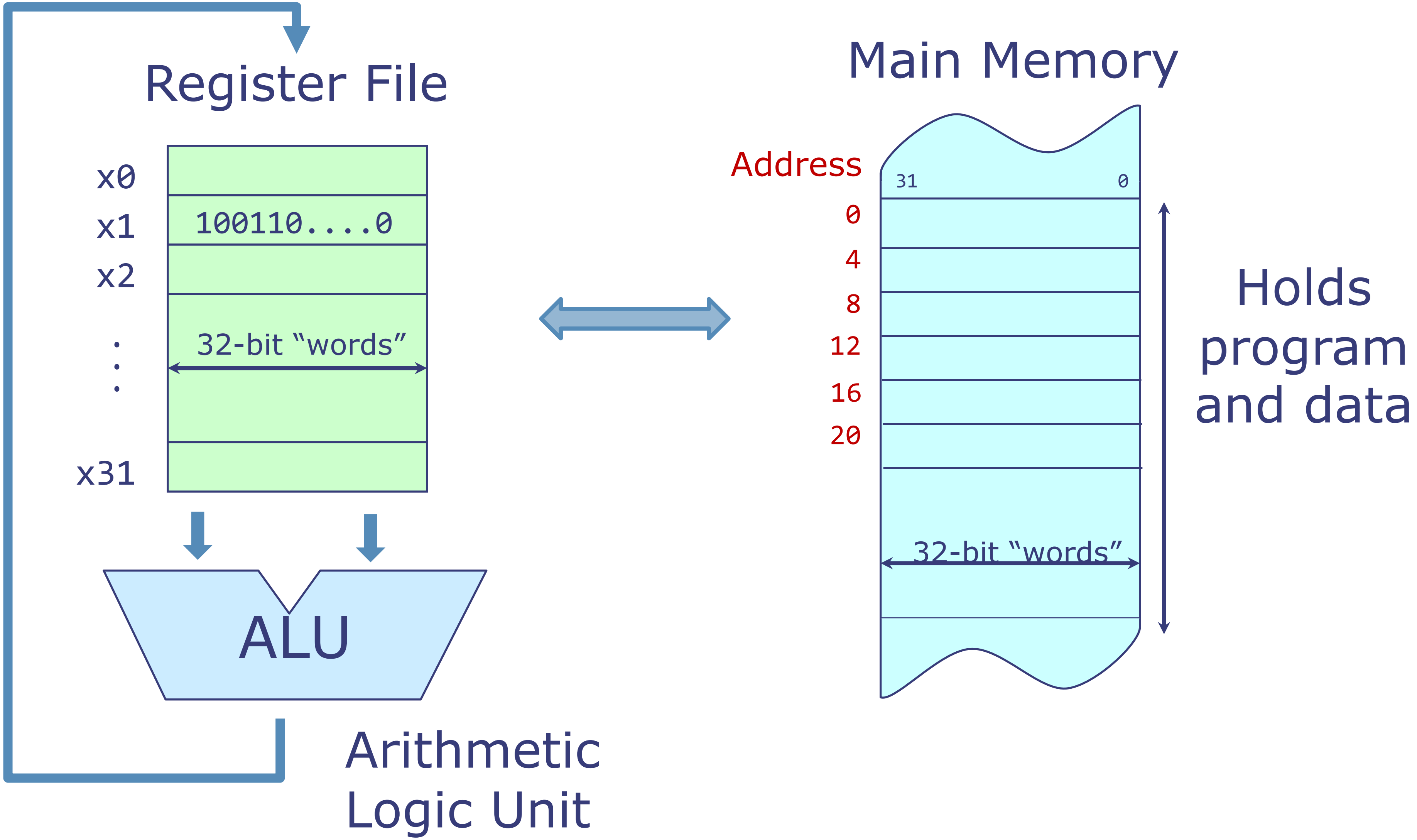


Components of a MicroProcessor





Components of a MicroProcessor



Register File

x0

x1 100110....0

x2

32-bit "words"

x31

ALU

Arithmetic Logic Unit

Main Memory

Address

31 0

0

4

8

12

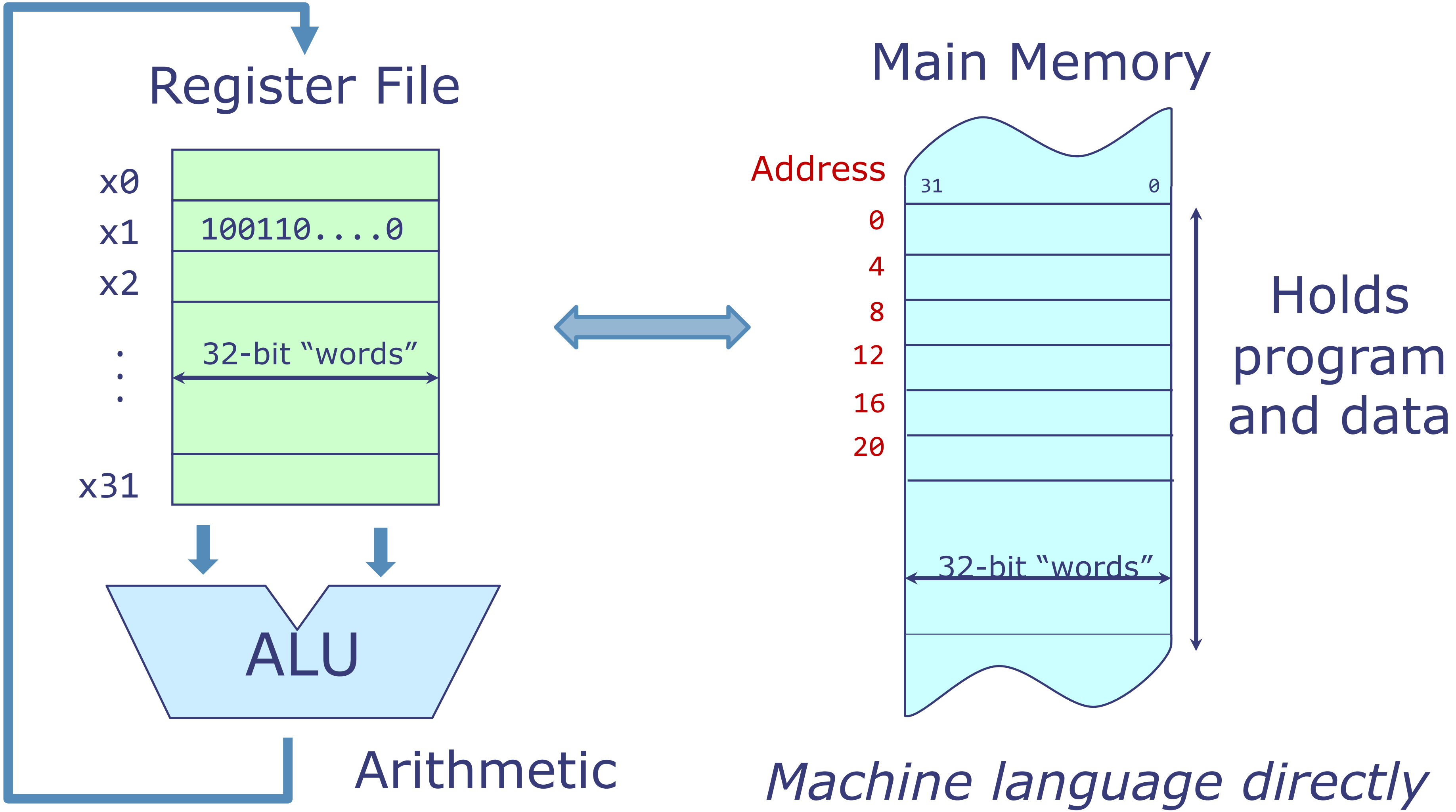
16

20

32-bit "words"

Holds program and data

Components of a MicroProcessor



Machine language directly reflects this structure

MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits

MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32

MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - $x_i \leftarrow \mathrm{Op}(x_j, x_k)$ where Op $\in$ {+, AND, OR, <, >, ...}

MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - $x_i \leftarrow \mathrm{Op}(\ x_j\ ,\ x_k\ )$ where $\mathrm{Op} \in \{+, \mathrm{AND}, \mathrm{OR}, <, >, \ldots\}$
- Memory is large, say Giga bytes, and holds program and data

MicroProcessor Structure / Assembly Language

- Each register is of fixed size, say 32 bits
- The number of registers are small, say 32
- ALU directly performs operations on the register file, typically
  - $x_i \leftarrow \mathrm{Op}(x_j, x_k)$ where Op $\in$ {+, AND, OR, <, >, ...}
- Memory is large, say Giga bytes, and holds program and data
- Data can be moved back and forth between Memory and Register File
  - Ld x M[addr]
  - St M[addr] x

Assembly (Machine) Language Program

- An assembly language program is a sequence of instructions which execute in a sequential order unless a control transfer instruction is executed

Assembly (Machine) Language Program

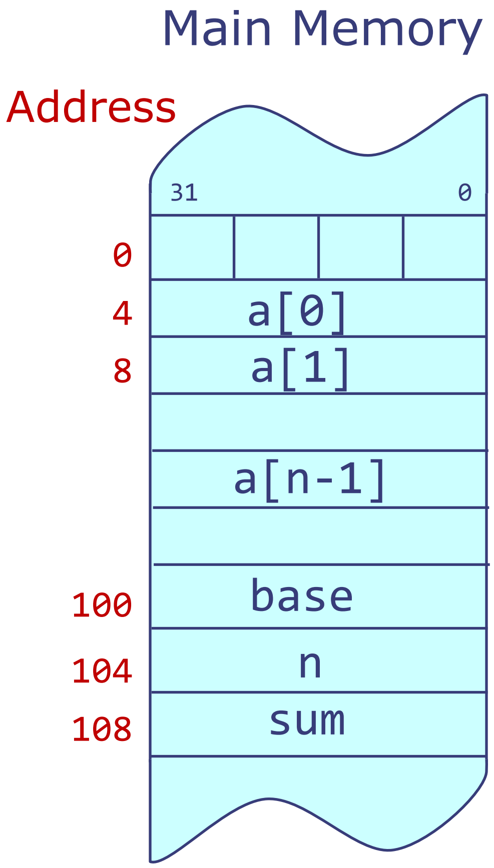
- An assembly language program is a sequence of instructions which execute in a sequential order unless a control transfer instruction is executed
- Each instruction specifies one of the following operations:
  - ALU or Reg-to-Reg operation
  - Ld
  - St
  - Control transfer operation: e.g., if xi < xj go to label l

Program to sum array elements

```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```

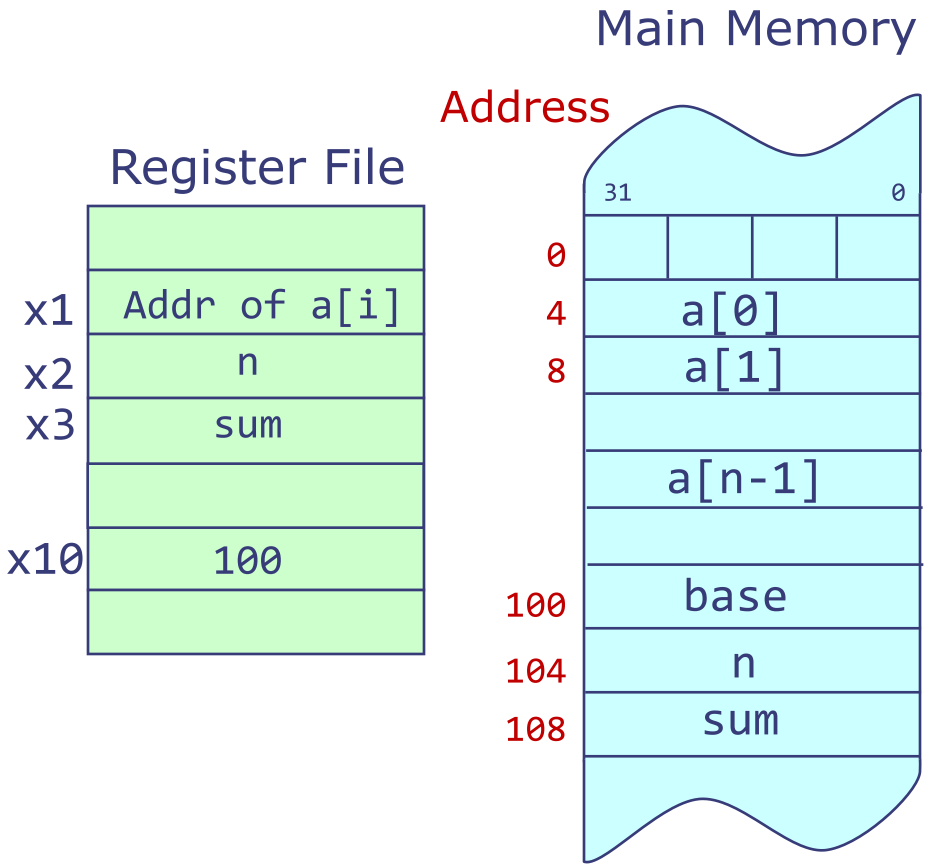
Program to sum array elements

```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```



Program to sum array elements

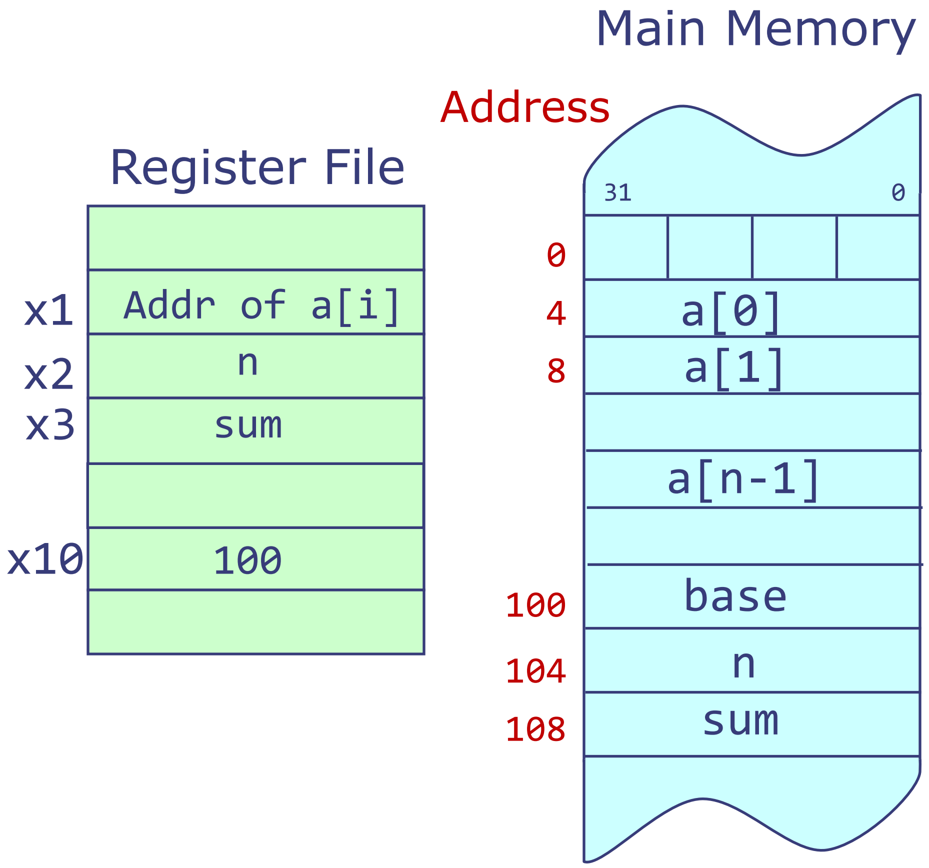
```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```



Program to sum array elements

```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```

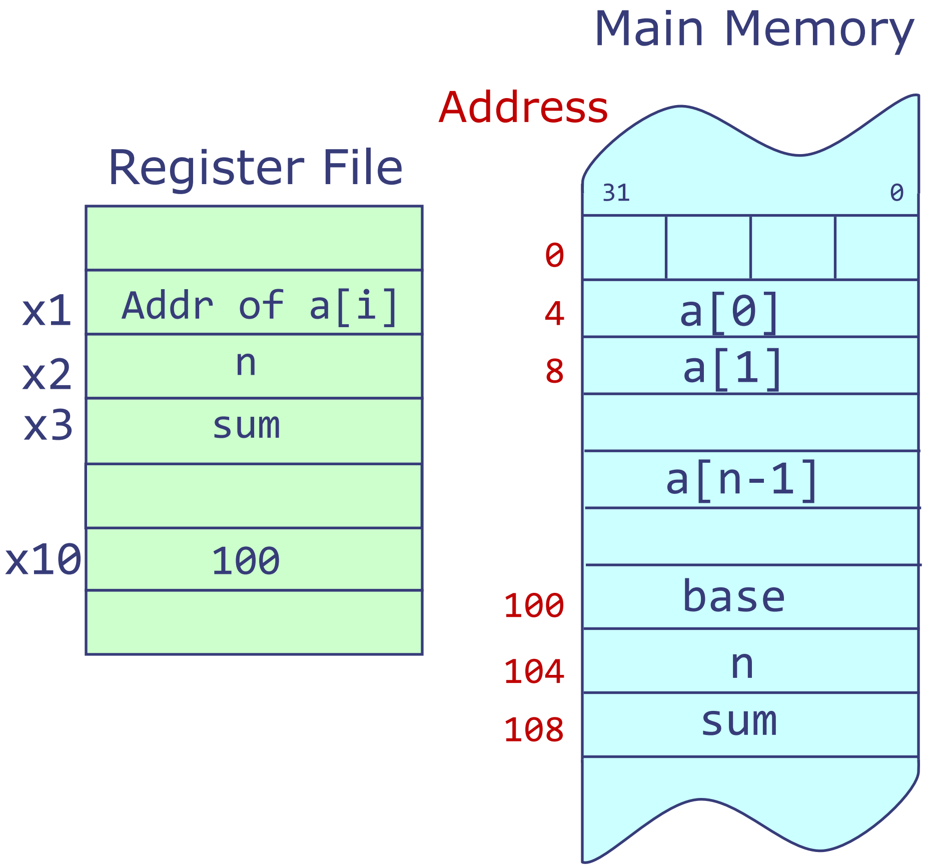
```
x1 ⍰ load(base)
x2 ⍰ load(n)
x3 ⍰ 0
```



Program to sum array elements

```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```

```
   x1 ⍰ load(base)
   x2 ⍰ load(n)
   x3 ⍰ 0
loop:
   x4 ⍰ load(Mem[x1])
   add x3, x3, x4
   addi x1, x1, 4
   addi x2, x2, -1
   bnez x2, loop
```

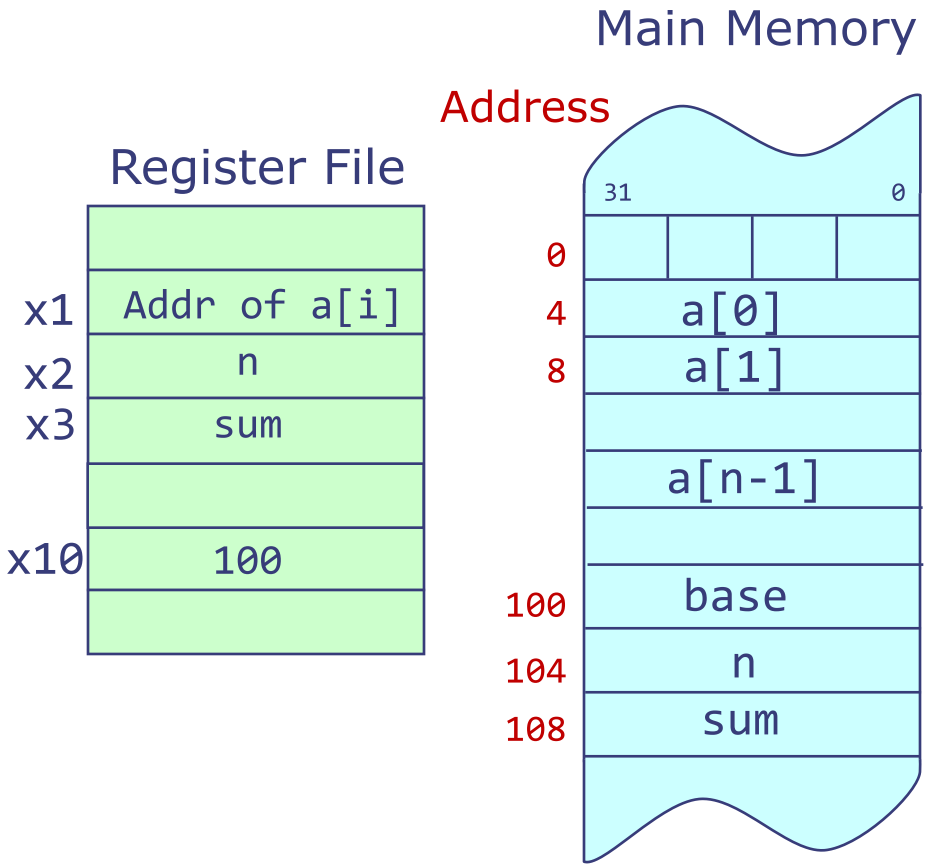


Program to sum array elements

```
sum = a[0] + a[1] + a[2] + ... + a[n-1]
```

```
    x1 ⍰ load(base)
    x2 ⍰ load(n)
    x3 ⍰ 0
loop:
    x4 ⍰ load(Mem[x1])
    add x3, x3, x4
    addi x1, x1, 4
    addi x2, x2, -1
    bnez x2, loop

    store(sum) ⍰ x3
```



High Level vs Assembly Language

High Level Language

Assembly Language

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations

Assembly Language

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations

Assembly Language

1. Primitive Arithmetic and logical operations

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures

Assembly Language

1. Primitive Arithmetic and logical operations

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers
3. Control transfer instructions

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures
4. Not suitable for direct implementation in hardware

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers
3. Control transfer instructions

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures
4. Not suitable for direct implementation in hardware

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers
3. Control transfer instructions
4. Designed to be directly implementable in hardware

High Level vs Assembly Language

High Level Language

1. Primitive Arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - Conditional statements, loops and procedures
4. Not suitable for direct implementation in hardware

Assembly Language

1. Primitive Arithmetic and logical operations
2. Primitive data structures – bits and integers
3. Control transfer instructions
4. Designed to be directly implementable in hardware

tedious programming!

Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

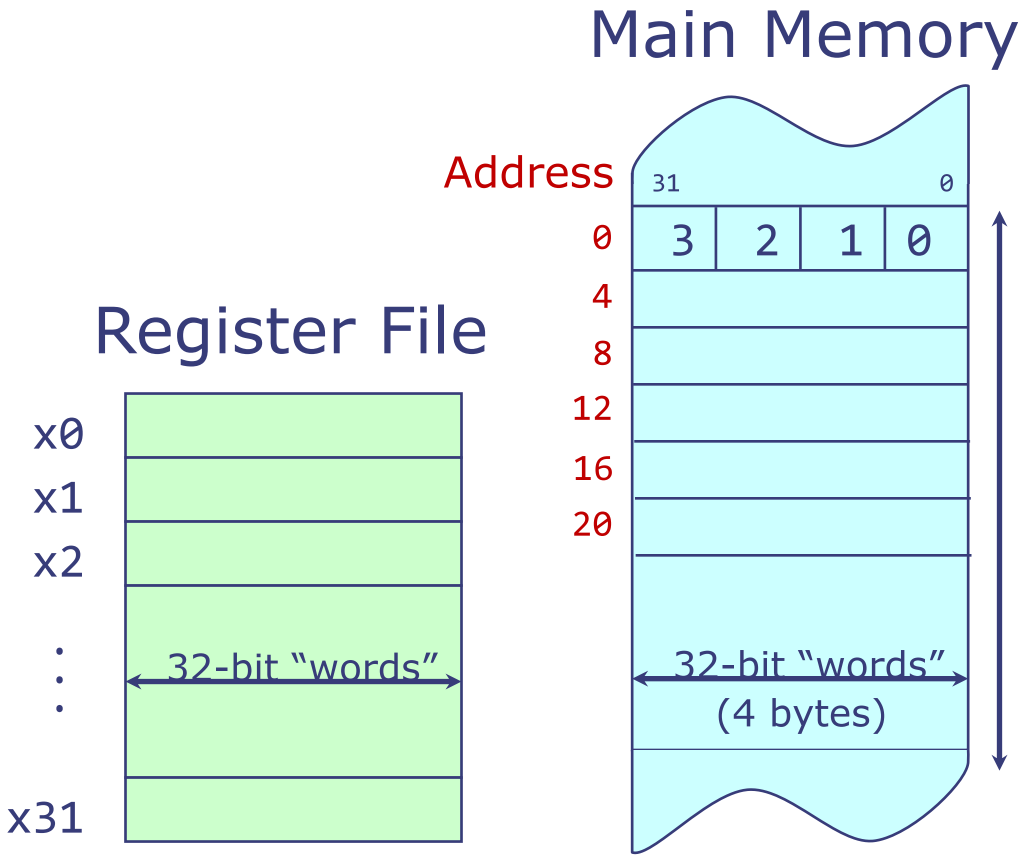
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
- RISC-V ISA:
  - A new, open, free ISA from Berkeley
  - Several variants
    - RV32, RV64, RV128: Different data widths
    - ‘I’: Base Integer instructions
    - ‘M’: Multiply and Divide
    - ‘F’ and ‘D’: Single- and Double-precision floating point
    - And many other modular extensions

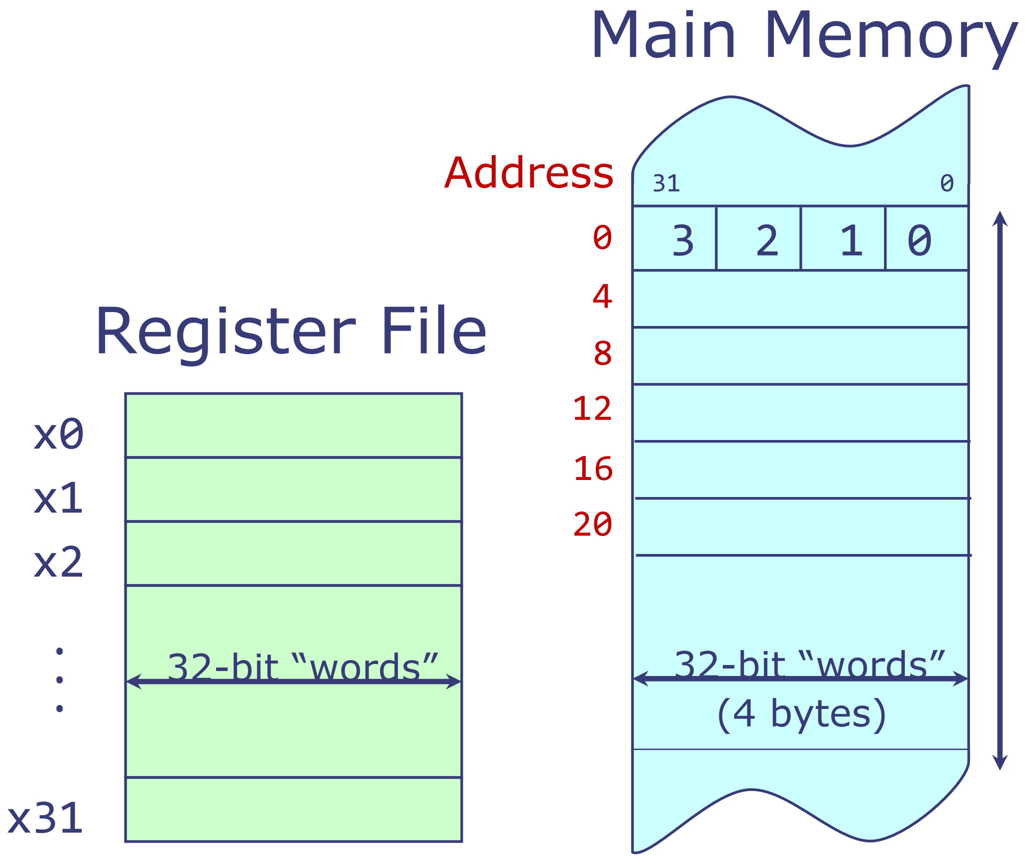
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
- RISC-V ISA:
  - A new, open, free ISA from Berkeley
  - Several variants
    - RV32, RV64, RV128: Different data widths
    - 'I': Base Integer instructions
    - 'M': Multiply and Divide
    - 'F' and 'D': Single- and Double-precision floating point
    - And many other modular extensions
- We will design an RV32I processor, which is the base integer 32-bit variant

RISC-V Processor Storage



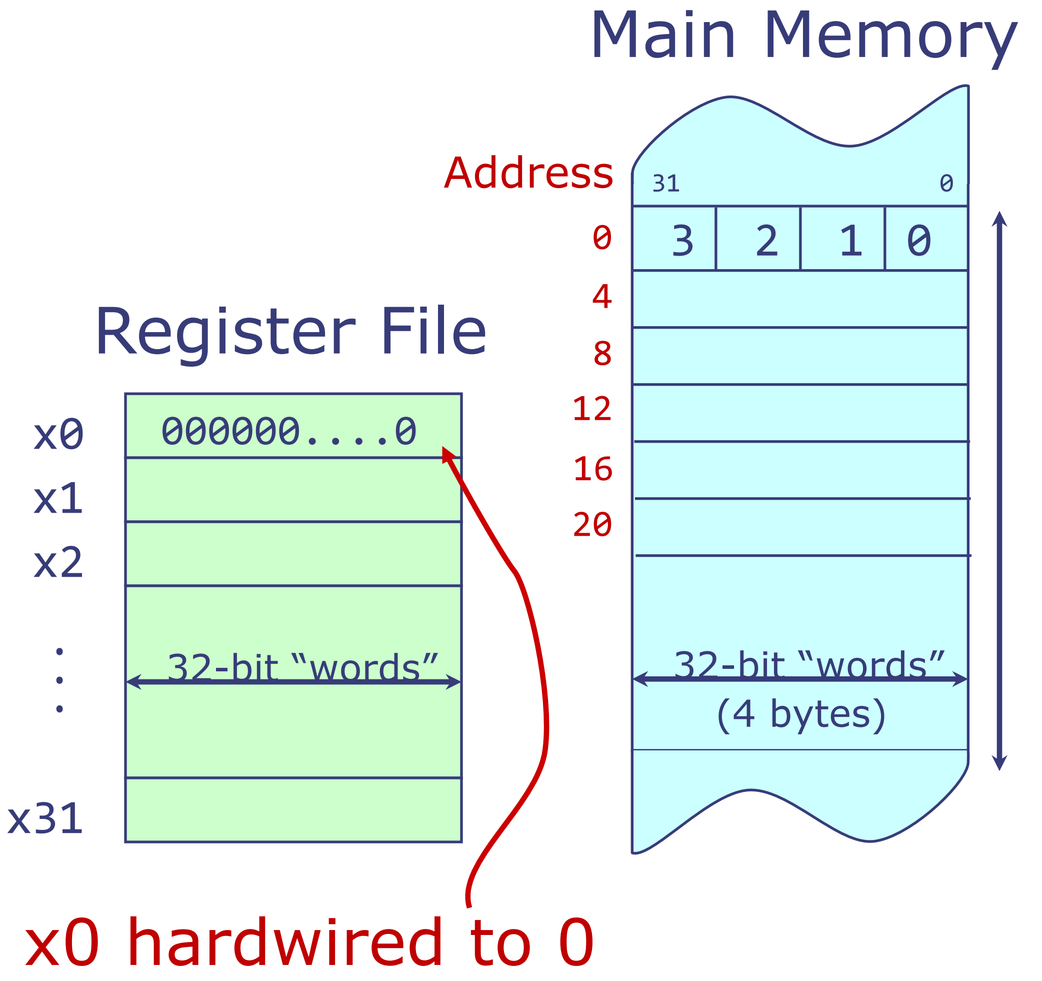
RISC-V Processor Storage



Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide

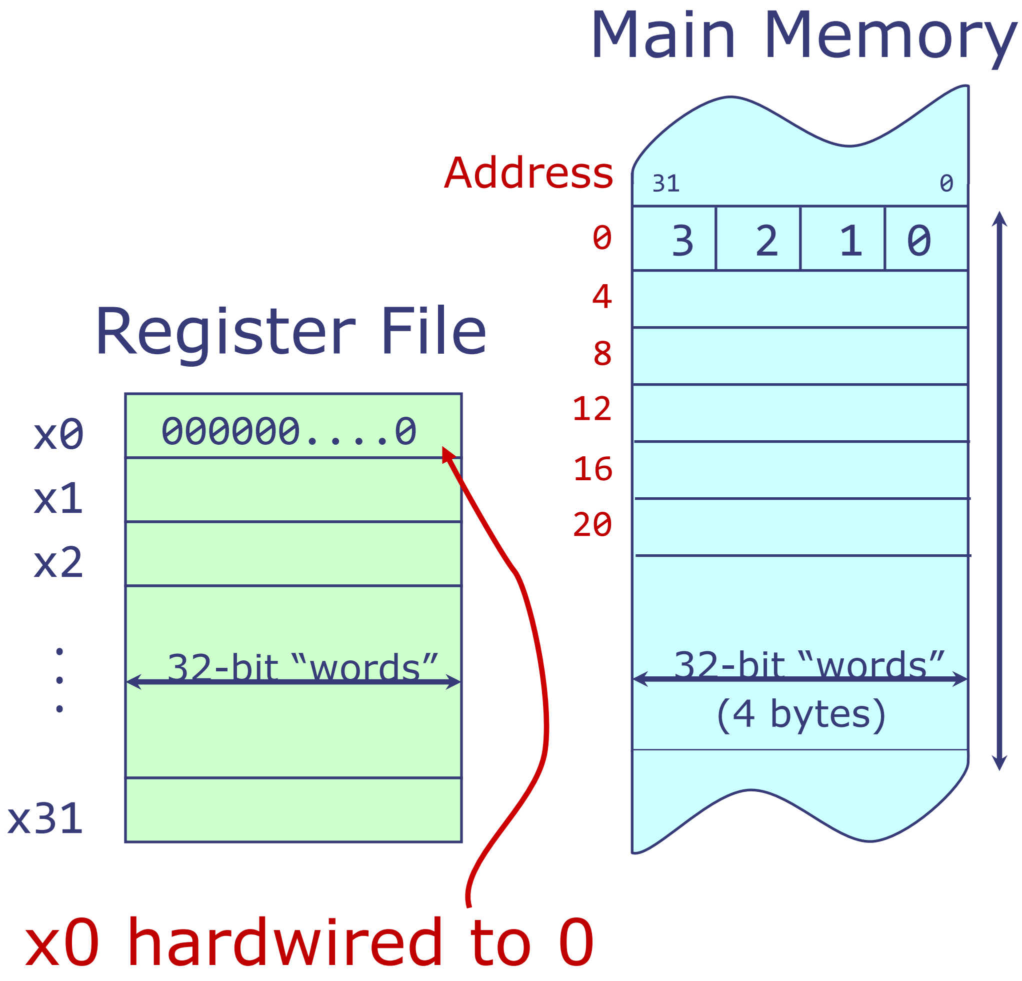
RISC-V Processor Storage



Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

RISC-V Processor Storage



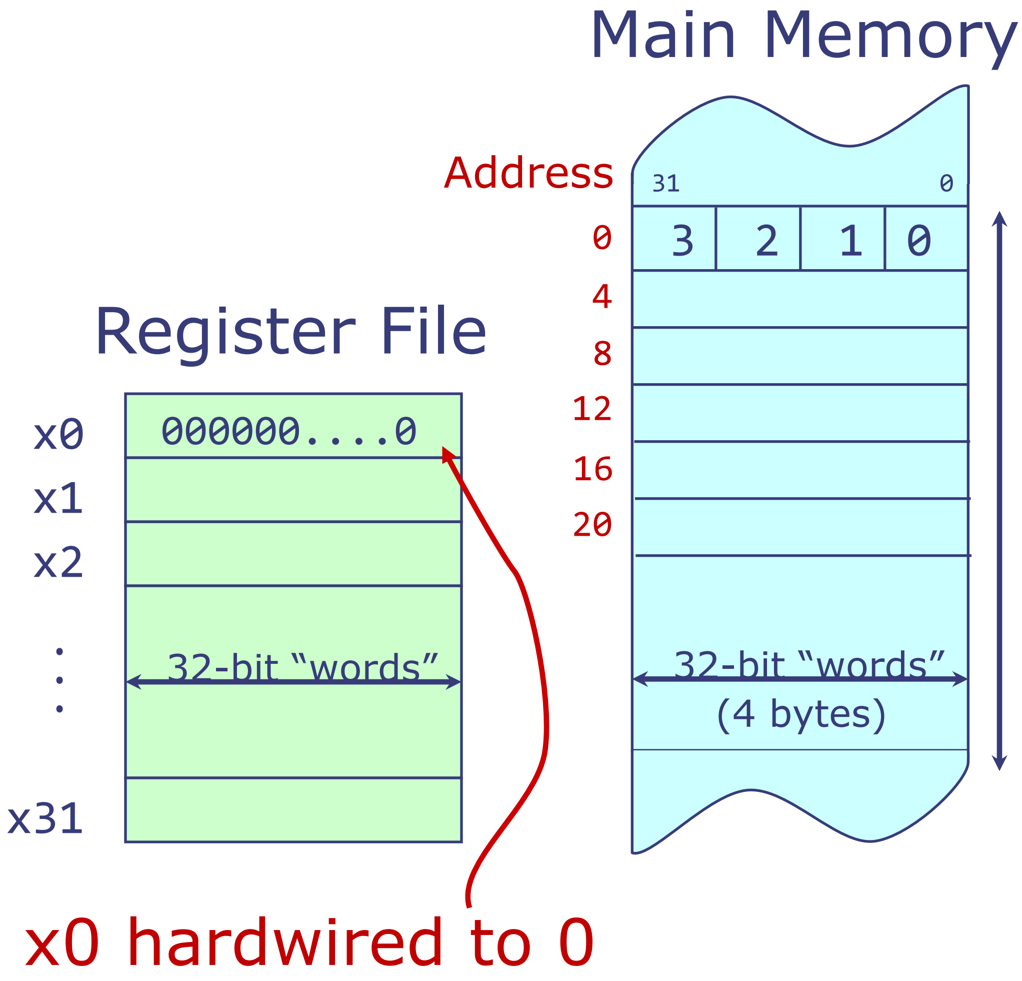
Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

Memory:

- Each memory location is 32 bits wide (1 word)
  - Instructions and data

RISC-V Processor Storage



x0 hardwired to 0

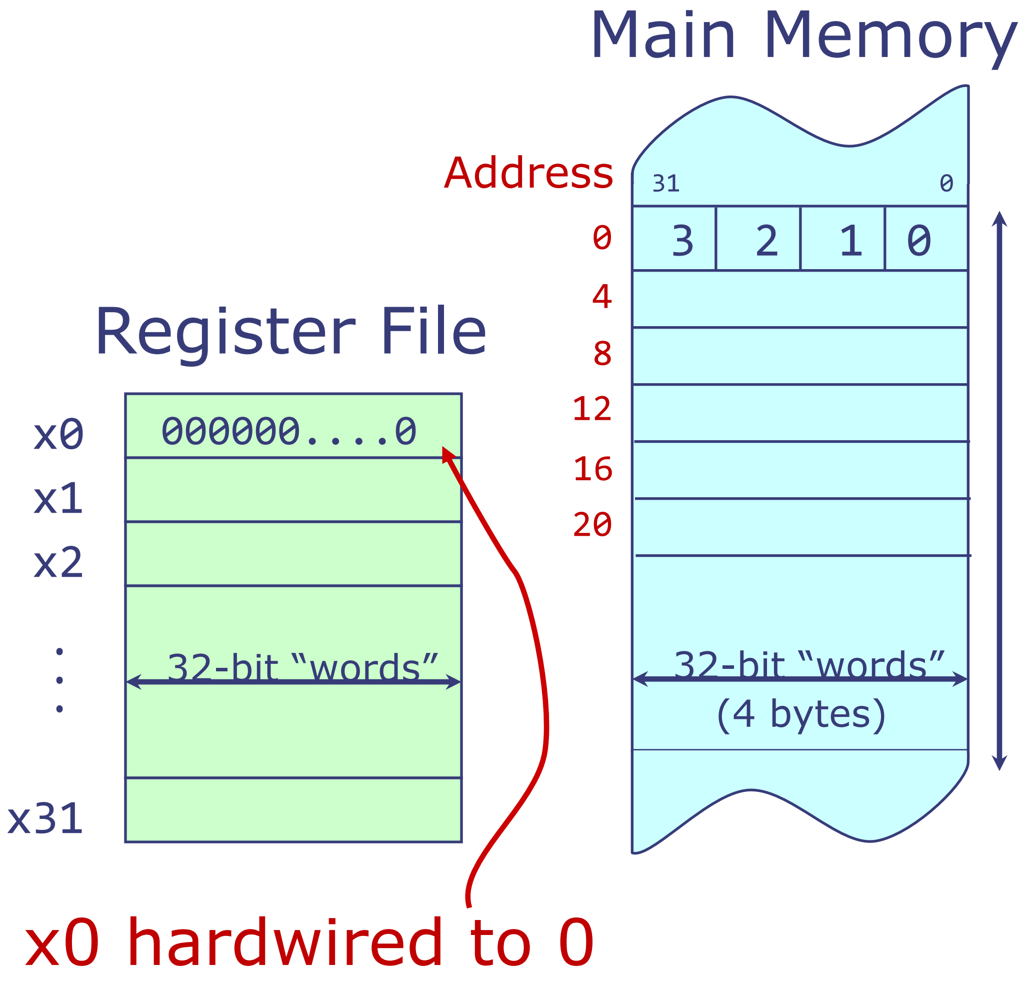
Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

Memory:

- Each memory location is 32 bits wide (1 word)
  - Instructions and data
- Memory is byte (8 bits) addressable
- Address of adjacent words are 4 apart.

RISC-V Processor Storage



Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide
- x0 = 0

Memory:

- Each memory location is 32 bits wide (1 word)
  - Instructions and data
- Memory is byte (8 bits) addressable
- Address of adjacent words are 4 apart.
- Address is 32 bits
- Can address 2^{32} bytes or 2^{30} words.

RISC-V ISA: Instructions

- Three types of operations:
  - **Computational:** Perform arithmetic and logical operations on registers
  - **Loads and stores:** Move data between registers and main memory
  - **Control Flow:** Change the execution order of instructions to support conditional statements and loops.

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
    - Register-Register Instructions:
        - 2 source operand registers
        - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

- **Format:** `oper dest, src1, src2`

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

  - **Format:** `oper dest, src1, src2`

- `add x3, x1, x2`
- `x3 ← x1 + x2`

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

- **Format:** `oper dest, src1, src2`

- `add x3, x1, x2`
- `slt x3, x1, x2`

- `x3 ← x1 + x2`
- `If x1 < x2 then x3 = 1 else x3 = 0`

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

- **Format:** `oper dest, src1, src2`

`add x3, x1, x2`	`x3 ← x1 + x2`
`slt x3, x1, x2`	`If x1 < x2 then x3 = 1 else x3 = 0`
`and x3, x1, x2`	`x3 ← x1 & x2`

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
  - Register-Register Instructions:
    - 2 source operand registers
    - 1 destination register

Arithmetic	Comparisons	Logical	Shifts
`add, sub`	`slt, sltu`	`and, or, xor`	`sll, srl, sra`

- **Format:** `oper dest, src1, src2`

`add x3, x1, x2`	`x3 ← x1 + x2`
`slt x3, x1, x2`	`If x1 < x2 then x3 = 1 else x3 = 0`
`and x3, x1, x2`	`x3 ← x1 & x2`
`sll x3, x1, x2`	`x3 ← x1 << x2`

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{r} 5 \\ +\quad 3 \\ \hline \mathit{8} \end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
5	00101
+ 3	+ 00011
8	

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{rr} & 5 \\ + & 3 \\ \hline & 8 \end{array}$$

Base 2

$$\begin{array}{rccccc} & & & & 1 & \\ & 0 & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 1 \\ \hline & & & & & 0 \end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{rr} & 5 \\ + & 3 \\ \hline & 8 \end{array}$$

Base 2

$$\begin{array}{rccccc} & & & 1 & 1 & \\ & 0 & 0 & 1 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 1 \\ \hline & & & & 0 & 0 \end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{rr} & 5 \\ + & 3 \\ \hline & 8 \end{array}$$

Base 2

$$\begin{array}{rr} & 111 \\ & 00101 \\ + & 00011 \\ \hline & 000 \end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{r} 5 \\ +\quad 3 \\ \hline 8 \end{array}$$

Base 2

$$\begin{array}{r} 111\ \ \\ 00101 \\ +\quad 00011 \\ \hline 1000 \end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10

$$\begin{array}{rr}
 & 5 \\
+ & 3 \\ \hline
 & 8
\end{array}$$

Base 2

$$\begin{array}{rr}
 & 111 \\
 & 00101 \\
+ & 00011 \\ \hline
 & 01000
\end{array}$$

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

- `sll x3, x1, x2`
  `Shift x1 left`
  `by x2 bits`

00101

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

  - `sll x3, x1, x2`

    `Shift x1 left`

    `by x2 bits`

00101

0101*0*

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

  - `sll x3, x1, x2`

    `Shift x1 left`

    `by x2 bits`

00101

0101*0*

101*00*

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

- `sll x3, x1, x2`

  `Shift x1 left`

  `by x2 bits`

00101
0101*0*
101*00*
01*000*

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - `add x3, x1, x2`

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

- `sll x3, x1, x2`

  `Shift x1 left`

  `by x2 bits`

00101
0101*0*
101*00*
01*000*

Notice fixed width

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format:** `oper dest, src1, const`

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format:** `oper dest, src1, const`
  - `addi x3, x1, 3` — `x3 ← x1 + 3`
  - `andi x3, x1, 3` — `x3 ← x1 & 3`
  - `slli x3, x1, 3` — `x3 ← x1 << 3`

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format:** `oper dest, src1, const`
  - `addi x3, x1, 3`
  - `andi x3, x1, 3`
  - `slli x3, x1, 3`
  - `x3 ← x1 + 3`
  - `x3 ← x1 & 3`
  - `x3 ← x1 << 3`

Format	Arithmetic	Comparisons	Logical	Shifts
Register-Register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
Register-Immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - **Format:** `oper dest, src1, const`
  - `addi x3, x1, 3` — x3 ← x1 + 3
  - `andi x3, x1, 3` — x3 ← x1 & 3
  - `slli x3, x1, 3` — x3 ← x1 << 3

Format	Arithmetic	Comparisons	Logical	Shifts
Register-Register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
Register-Immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

- No subi, instead use negative constant.
  - `addi x3, x1, -3` — x3 ← x1 - 3

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

1. Break up complex expression into basic computations.

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

1. Break up complex expression into basic computations.
    - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

```
t0 = b + 3;
t1 = t0 >> c;
a = t1 - 1;
```

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

1. Break up complex expression into basic computations.
    - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

2. Assume a, b, c are in registers x1, x2, and x3 respectively. Use x4 for t0, and x5 for t1.

```
t0 = b + 3;
t1 = t0 >> c;
a = t1 - 1;
```

Compound Computation

- Execute `a = ((b+3) >> c) - 1;`

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand (also known as three address instruction)

2. Assume a, b, c are in registers x1, x2, and x3 respectively. Use x4 for t0, and x5 for t1.

```
t0 = b + 3;
t1 = t0 >> c;
a = t1 - 1;
```

```
addi x4, x2, 3
srl x5, x4, x3
addi x1, x5, -1
```

Control Flow Instructions

- Execute

  ```
  if (a < b):   c = a + 1
        else:   c = b + 2
  ```

Control Flow Instructions

- Execute

```
if (a < b):   c = a + 1
      else:   c = b + 2
```

- Need Conditional branch instructions:
  - Format: `comp src1, src2, label`

Control Flow Instructions

- Execute `if (a < b):    c = a + 1`
  `else:    c = b + 2`
- Need Conditional branch instructions:
  - Format: `comp src1, src2, label`
  - First performs comparison to determine if branch is taken or not: `src1` *`comp`* `src2`

Control Flow Instructions

- Execute

  ```
  if (a < b):   c = a + 1
        else:   c = b + 2
  ```

- Need Conditional branch instructions:
  - Format: `comp src1, src2, label`
  - First performs comparison to determine if branch is taken or not: `src1` *`comp`* `src2`
  - If comparison returns True, then branch is taken, else continue executing program in order.

Control Flow Instructions

- Execute `if (a < b):   c = a + 1`
  `                else:   c = b + 2`
- Need Conditional branch instructions:
  - Format: `comp src1, src2, label`
  - First performs comparison to determine if branch is taken or not: `src1` *`comp`* `src2`
  - If comparison returns True, then branch is taken, else continue executing program in order.

Instruction	beq	bne	blt	bge	bltu	bgeu
comp	==	!=	<	≥	<	≥

Control Flow Instructions

- Execute
  ```
  if (a < b):   c = a + 1
        else:   c = b + 2
  ```
- Need Conditional branch instructions:
  - Format: `comp src1, src2, label`
  - First performs comparison to determine if branch is taken or not: `src1` *`comp`* `src2`
  - If comparison returns True, then branch is taken, else continue executing program in order.

Instruction	beq	bne	blt	bge	bltu	bgeu
comp	==	!=	<	≥	<	≥

```
       bge x1, x2, else
       addi x3, x1, 1
       beq x0, x0, end
else:  addi x3, x2, 2
end:
```

Assume
`x1=a; x2=b; x3=c;`

Unconditional Control Instructions: Jumps

- jal: Unconditional jump and link
  - Example: `jal x3, label`
  - Jump target specified as label
  - label is encoded as an offset from current instruction
  - Link (To be discussed next lecture): is stored in x3

Unconditional Control Instructions: Jumps

- jal: Unconditional jump and link
  - Example: `jal x3, label`
  - Jump target specified as label
  - label is encoded as an offset from current instruction
  - Link (To be discussed next lecture): is stored in x3
- jalr: Unconditional jump via register and link
  - Example: `jalr x3, 4(x1)`
  - Jump target specified as register value plus constant offset
  - Example: Jump target = x1 + 4
  - Can jump to any 32 bit address – supports long jumps

Constants and Instruction Encoding Limitations

- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a **small** constant.
  - Need to specify 1 destination register (5 bits).

Constants and Instruction Encoding Limitations

- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a **small** constant.
  - Need to specify 1 destination register (5 bits).

- The constant in register-immediate instructions has to be smaller than 12 bits; bigger constants have to be stored in the memory or a register and then used explicitly

Constants and Instruction Encoding Limitations

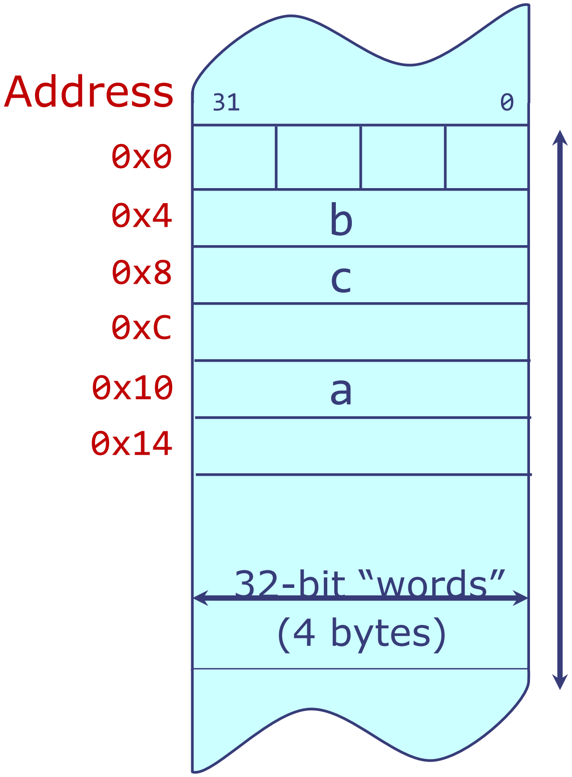
- Instructions are encoded as 32 bits.
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a **small** constant.
  - Need to specify 1 destination register (5 bits).

- The constant in register-immediate instructions has to be smaller than 12 bits; bigger constants have to be stored in the memory or a register and then used explicitly
- The constant in a jal instruction is 20 bits wide (7 bits for operation, and 5 bits for register)

Performing Computations on Values in Memory

a = b + c

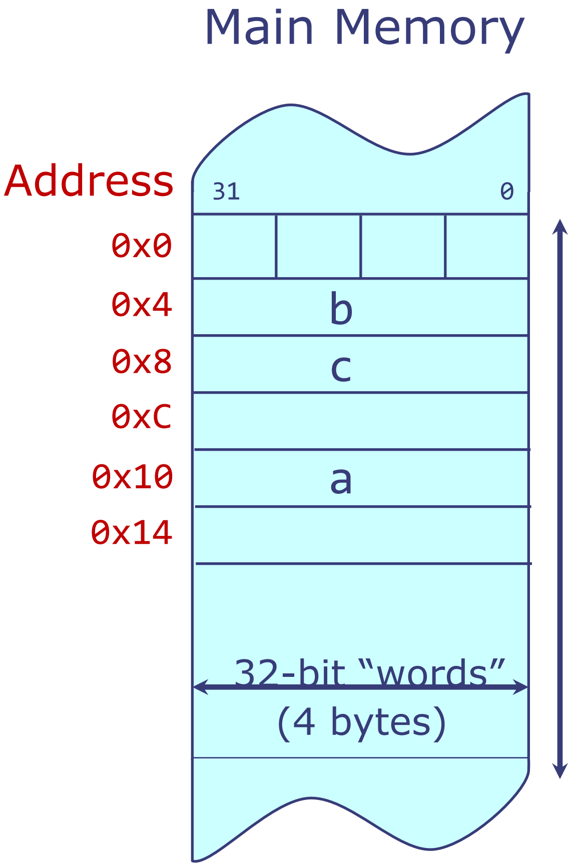
Main Memory



Performing Computations on Values in Memory

a = b + c

```
x1 ← load(Mem[b])
x2 ← load(Mem[c])
x3 ← x1 + x2
store(Mem[a]) ← x3
```

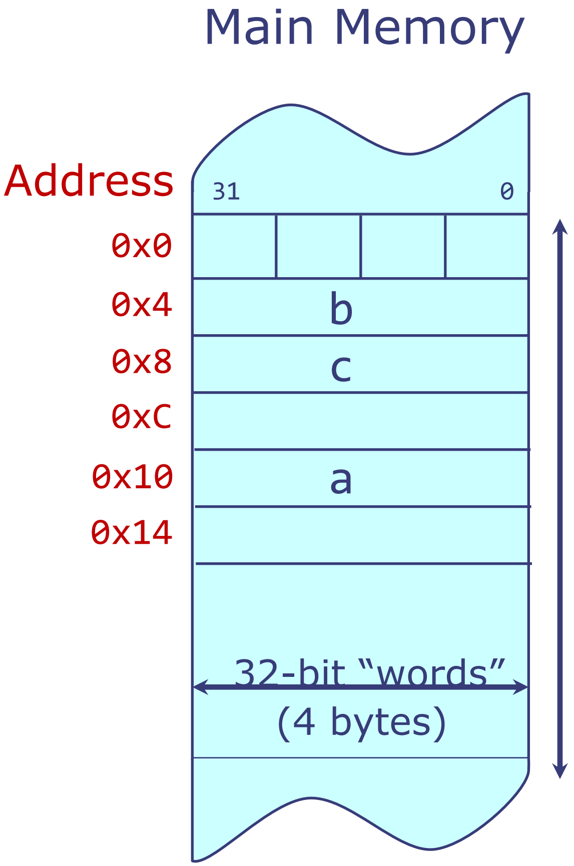


Performing Computations on Values in Memory

```
a = b + c

x1 ← load(Mem[b])
x2 ← load(Mem[c])
x3 ← x1 + x2
store(Mem[a]) ← x3
```

```
x1 ← load(0x4)
x2 ← load(0x8)
x3 ← x1 + x2
store(0x10) ← x3
```



RISC-V Load and Store Instructions

- Address is specified as a <base address, offset> pair;
  - base address is always stored in a register
  - the offset is specified as a small constant
  - Format: lw dest, offset(base)   sw src, offset(base)

RISC-V Load and Store Instructions

- Address is specified as a <base address, offset> pair;
  - base address is always stored in a register
  - the offset is specified as a small constant
  - Format: lw dest, offset(base)   sw src, offset(base)

- Assembly:

```
lw x1, 0x4(x0)
lw x2, 0x8(x0)
add x3, x1, x2
sw x3, 0x10(x0)
```

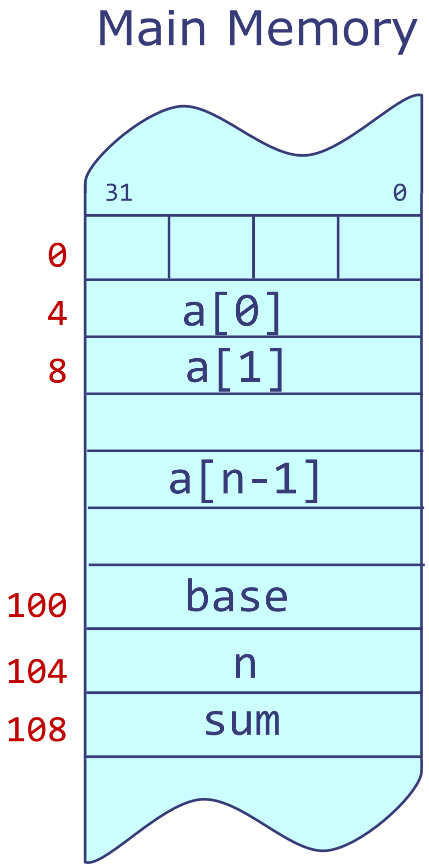
- Behavior:

```
x1 ← load(Mem[x0 + 0x4])
x2 ← load(Mem[x0 + 0x8])
x3 ← x1 + x2
store(Mem[x0 + 0x10]) ← x3
```

Program to sum array elements

`sum = a[0] + a[1] + a[2] + ... + a[n-1]`

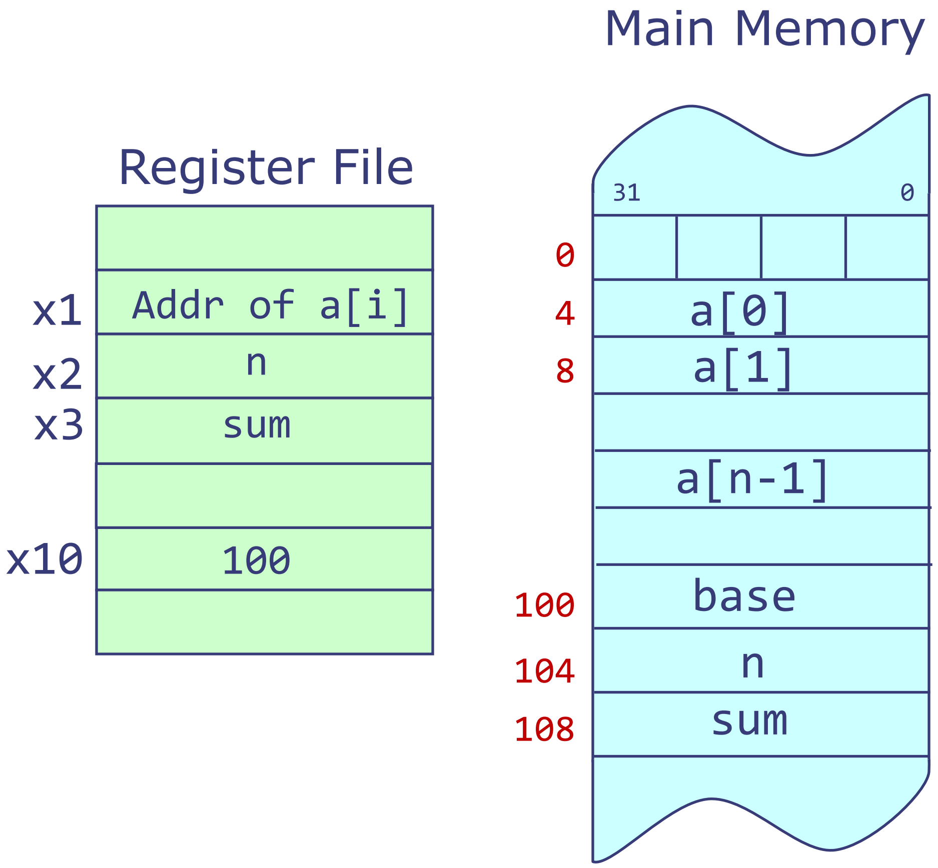
(Assume 100 (address of base) already loaded into x10)



Program to sum array elements

`sum = a[0] + a[1] + a[2] + ... + a[n-1]`

(Assume 100 (address of base) already loaded into x10)

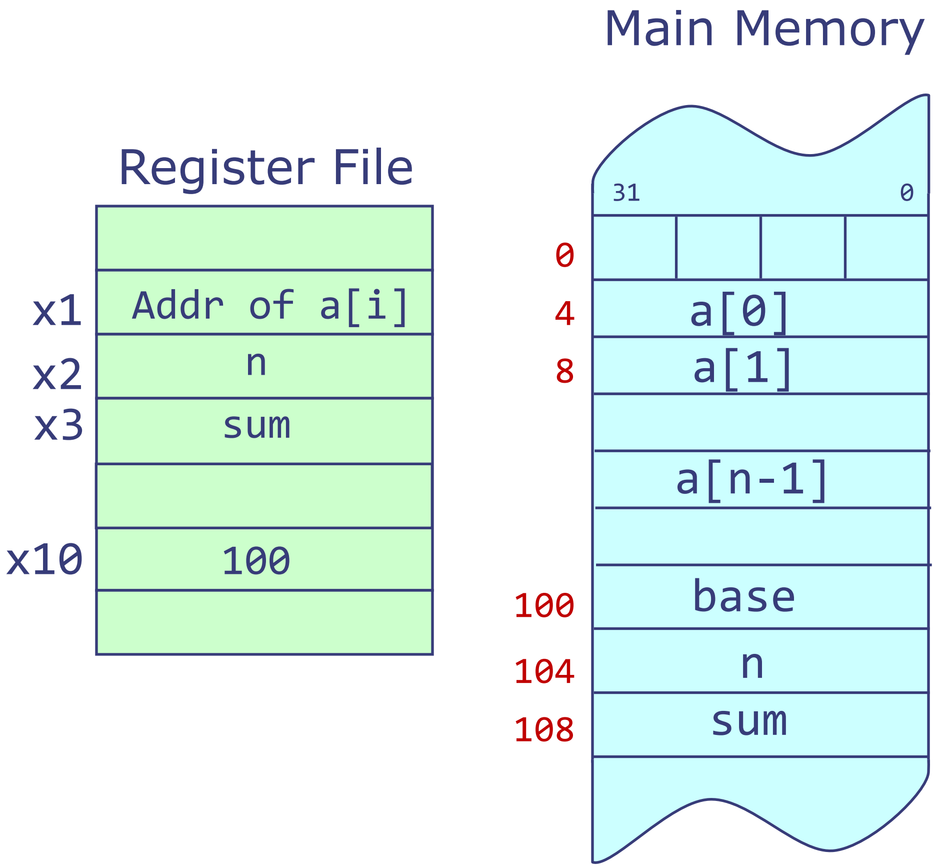


Program to sum array elements

`sum = a[0] + a[1] + a[2] + ... + a[n-1]`

(Assume 100 (address of base) already loaded into x10)

```
lw x1, 0x0(x10)
```

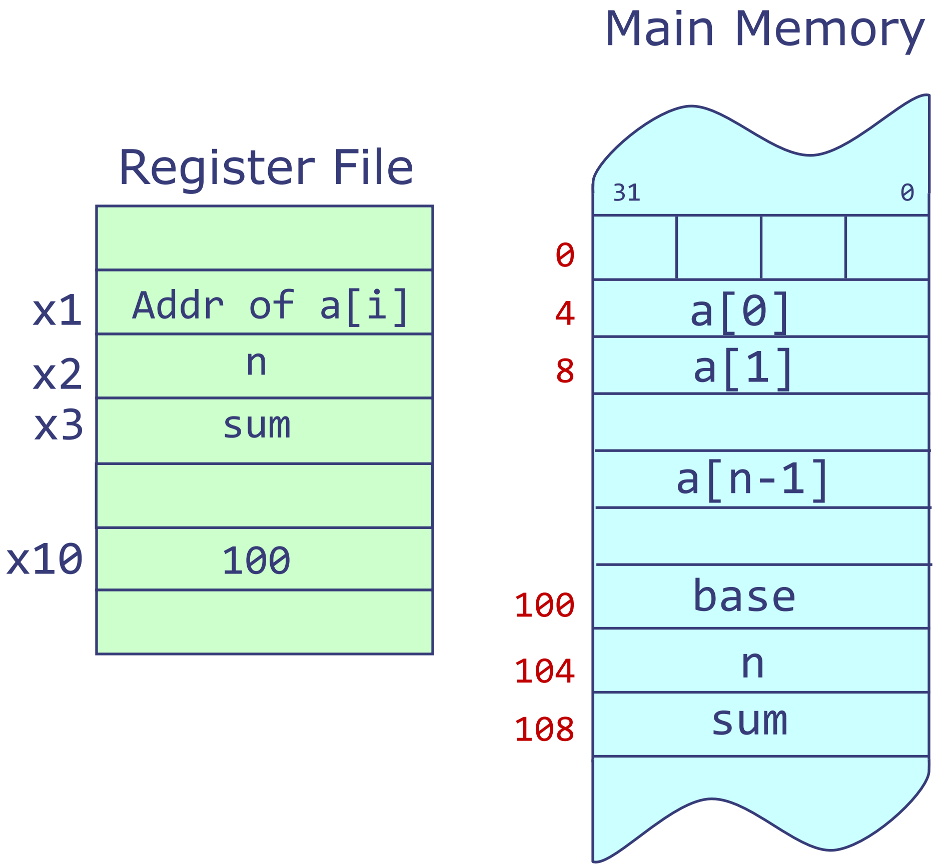


Program to sum array elements

`sum = a[0] + a[1] + a[2] + ... + a[n-1]`

(Assume 100 (address of base) already loaded into x10)

```
lw x1, 0x0(x10)
lw x2, 0x4(x10)
```

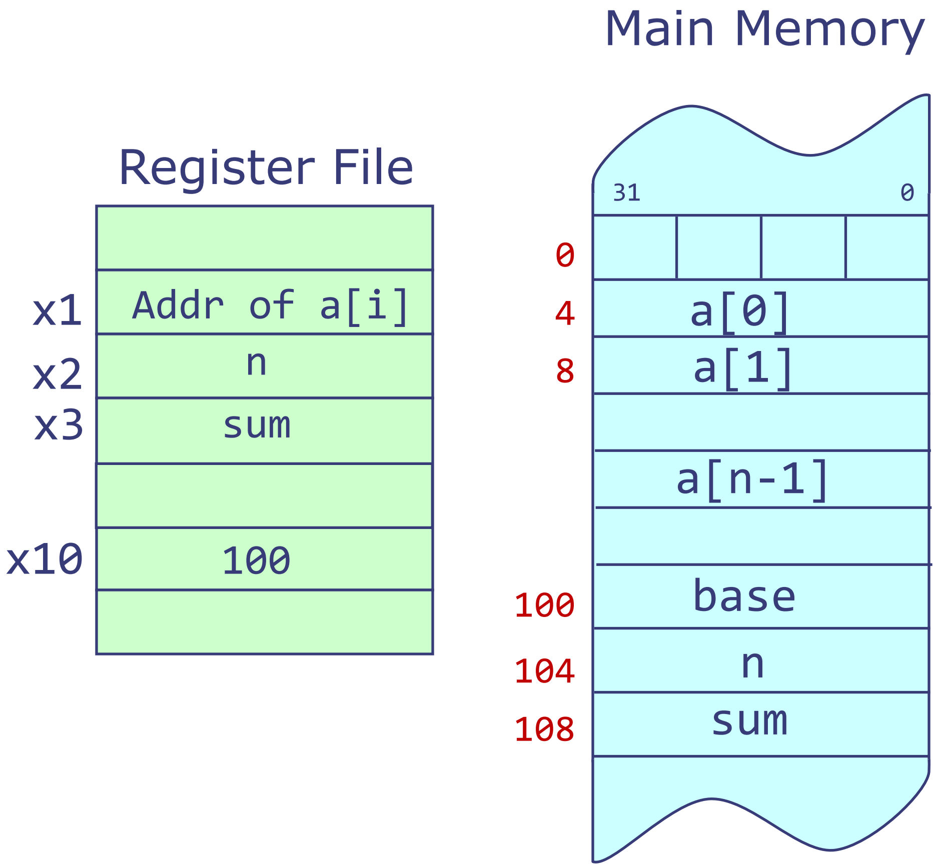


Program to sum array elements

sum = a[0] + a[1] + a[2] + ... + a[n-1]

(Assume 100 (address of base) already loaded into x10)

```
lw x1, 0x0(x10)
lw x2, 0x4(x10)
add x3, x0, x0
```

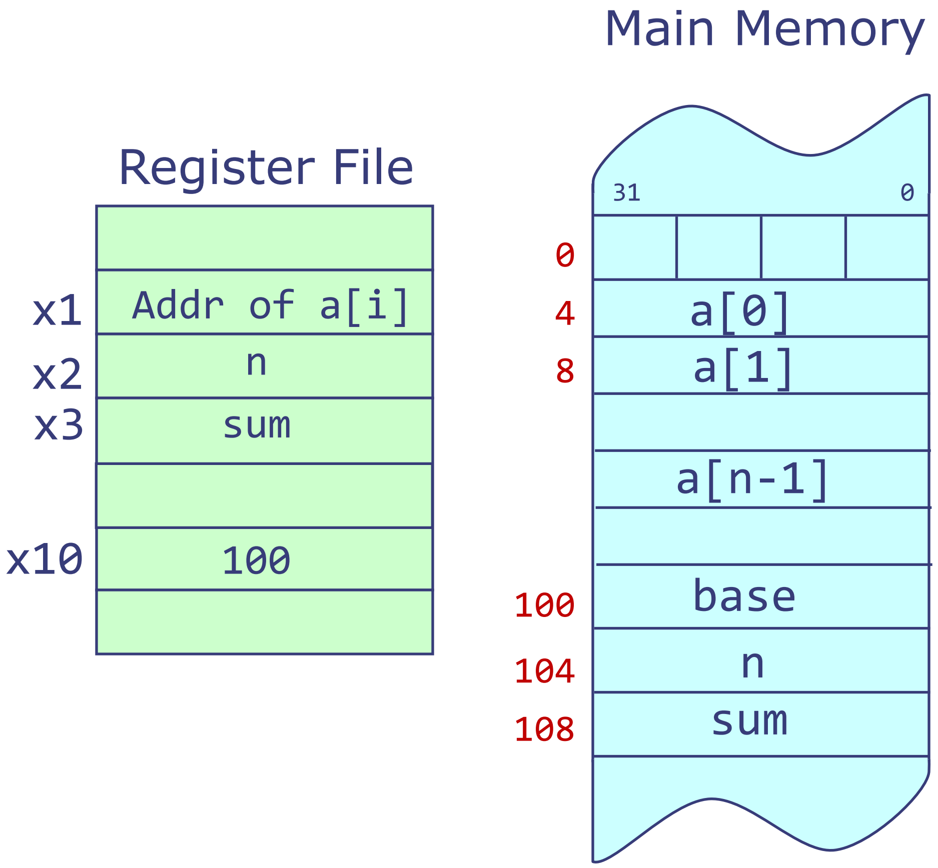


Program to sum array elements

sum = a[0] + a[1] + a[2] + ... + a[n-1]

(Assume 100 (address of base) already loaded into x10)

```
   lw x1, 0x0(x10)
   lw x2, 0x4(x10)
   add x3, x0, x0
loop:
   lw x4, 0x0(x1)
   add x3, x3, x4
   addi x1, x1, 4
   addi x2, x2, -1
   bnez x2, loop
```



Register File

x1	Addr of a[i]
x2	n
x3	sum
x10	100

Main Memory

	31 … 0
0	
4	a[0]
8	a[1]
	a[n-1]
100	base
104	n
108	sum

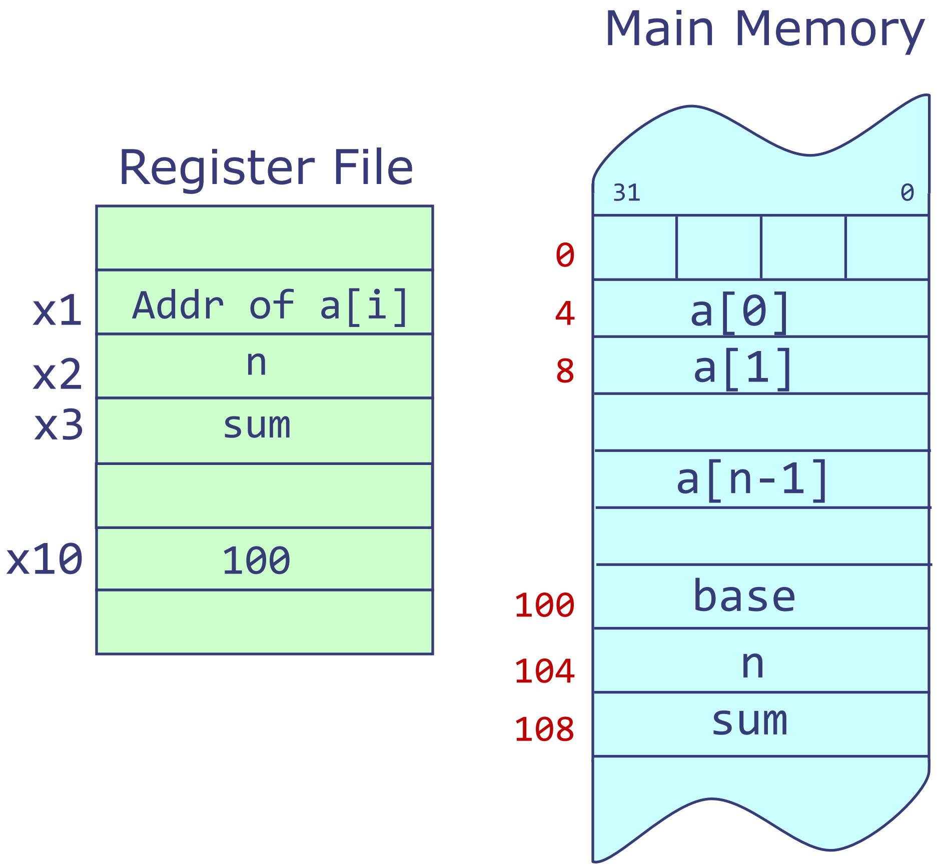
Program to sum array elements

sum = a[0] + a[1] + a[2] + ... + a[n-1]

(Assume 100 (address of base) already loaded into x10)

```
   lw x1, 0x0(x10)
   lw x2, 0x4(x10)
   add x3, x0, x0
loop:
   lw x4, 0x0(x1)
   add x3, x3, x4
   addi x1, x1, 4
   addi x2, x2, -1
   bnez x2, loop

   sw x3, 0x8(x10)
```



Register File

x1 Addr of a[i]

x2 n

x3 sum

x10 100

Main Memory

31 0

0

4 a[0]

8 a[1]

a[n-1]

100 base

104 n

108 sum

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction:	Equivalent Assembly Instruction:
`mv x2, x1`	`addi x2, x1, 0`

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction:	Equivalent Assembly Instruction:
mv x2, x1	addi x2, x1, 0
li x2, 3	addi x2, x0, 3

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction:	Equivalent Assembly Instruction:
`mv x2, x1`	`addi x2, x1, 0`
`li x2, 3`	`addi x2, x0, 3`
`ble x1, x2, label`	`bge x2, x1, label`

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

Pseudoinstruction:	Equivalent Assembly Instruction:
`mv x2, x1`	`addi x2, x1, 0`
`li x2, 3`	`addi x2, x0, 3`
`ble x1, x2, label`	`bge x2, x1, label`
`beqz x1, label`	`beq x1, x0, label`
`bnez x1, label`	`bne x1, x0, label`
`j label`	`jal x0, label`

Thank you!

Next lecture:
Implementing Procedures in Assembly