ECEC-355: Computer Organization & Architecture

Dr. Anup K. Das

Electrical and Computer Engineering

Drexel University

Content

- **Based on Chapter 2 of P&H**

Instruction Set Architecture

- **The repertoire of instructions of a computer**
- **Different computers have different instruction sets**
  - But with many aspects in common
- **Early computers had very simple instruction sets**
  - Simplified implementation
- **Many modern computers also have simple instruction sets**

The RISC-V ISA

- **Used as the example throughout the course**
- **Developed at UC Berkeley as open ISA**
- **Now managed by the RISC-V Foundation (riscv.org)**
- **Typical of many modern ISAs**
  - See RISC-V Reference Data tear-out card (book)
- **Similar ISAs have a large share of embedded core market**
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Arithmetic Operations

- **Add and subtract, three operands**
  - Two sources and one destination

```
add a, b, c  // a gets b + c
```

- **All arithmetic operations have this form**
- ***Design Principle 1:* Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- **C code:**

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

- **Compiled RISC-V code:**

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
sub f, t0, t1  // f = t0 - t1
```

- **Any operation must be carried out on the registers (t0, t1, g, h, i, j, f are all stored in registers)**

RISC-V Operations

- **Register Operations**
- **Memory Operations**
- **Immediate Operations**

Register Operands

- **Arithmetic instructions use register operands**
- **RISC-V has a 32 × 64-bit register file**
  - Use for frequently accessed data
  - 32-bit data is called a “word”
  - 64-bit data is called a “doubleword”
    - 32 x 64-bit general purpose registers x0 to x31
- ***Design Principle 2:* Smaller is faster**
  - c.f. main memory: millions of locations while registers are only 32
  - Registers are faster to access than memory
  - Reduced execution time of your code

RISC-V Registers

- 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

Register Operand Example

- **C code:**

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

  - f, …, j in x19, x20, …, x23

- **RISC-V code**

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
sub f, t0, t1  // f = t0 - t1
```

- **Compiled RISC-V code:**

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

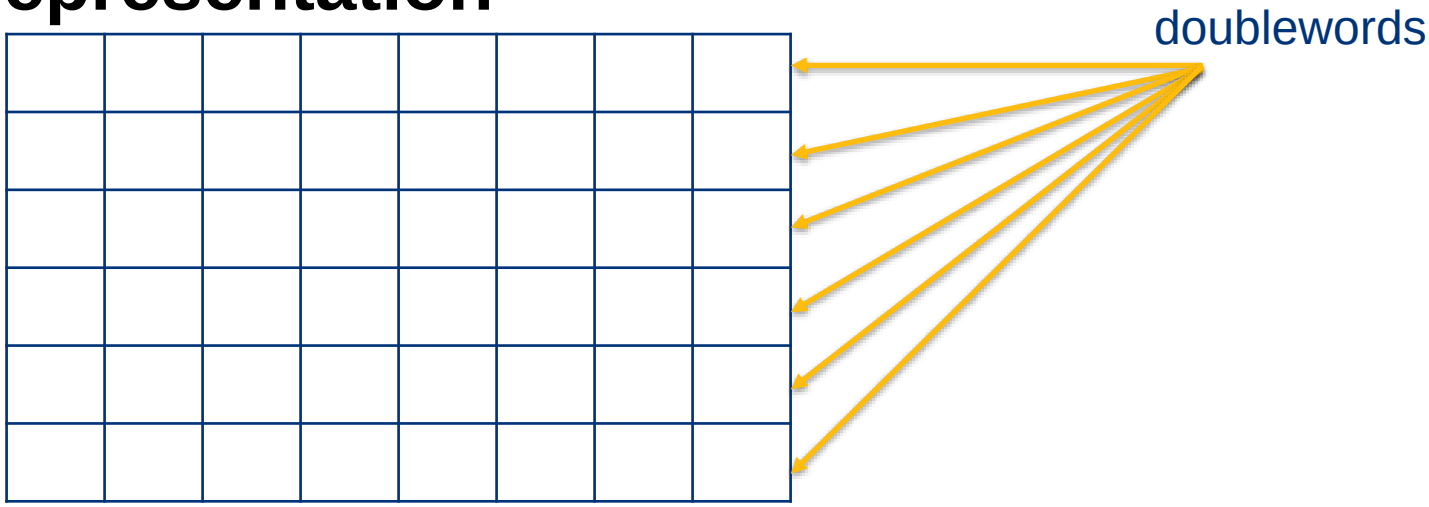
RISC-V Operations

- Register Operations
- **Memory Operations**
- **Immediate Operations**

Memory Operands

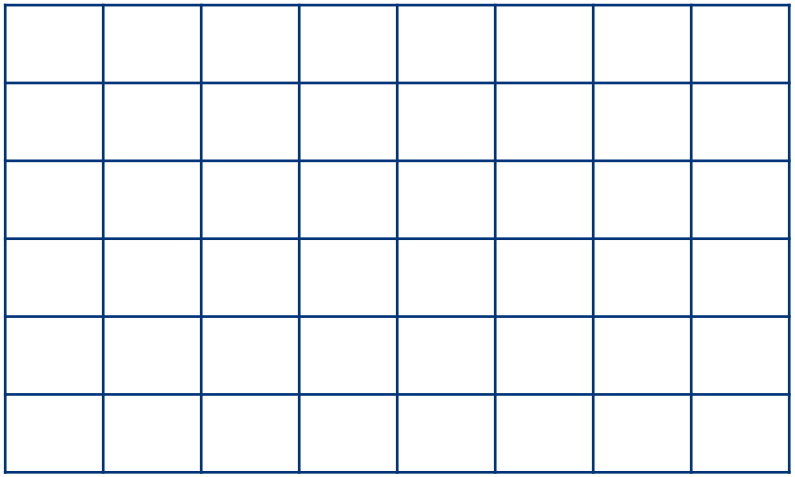
- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into registers
  - Store result from register to memory

- **Memory representation**



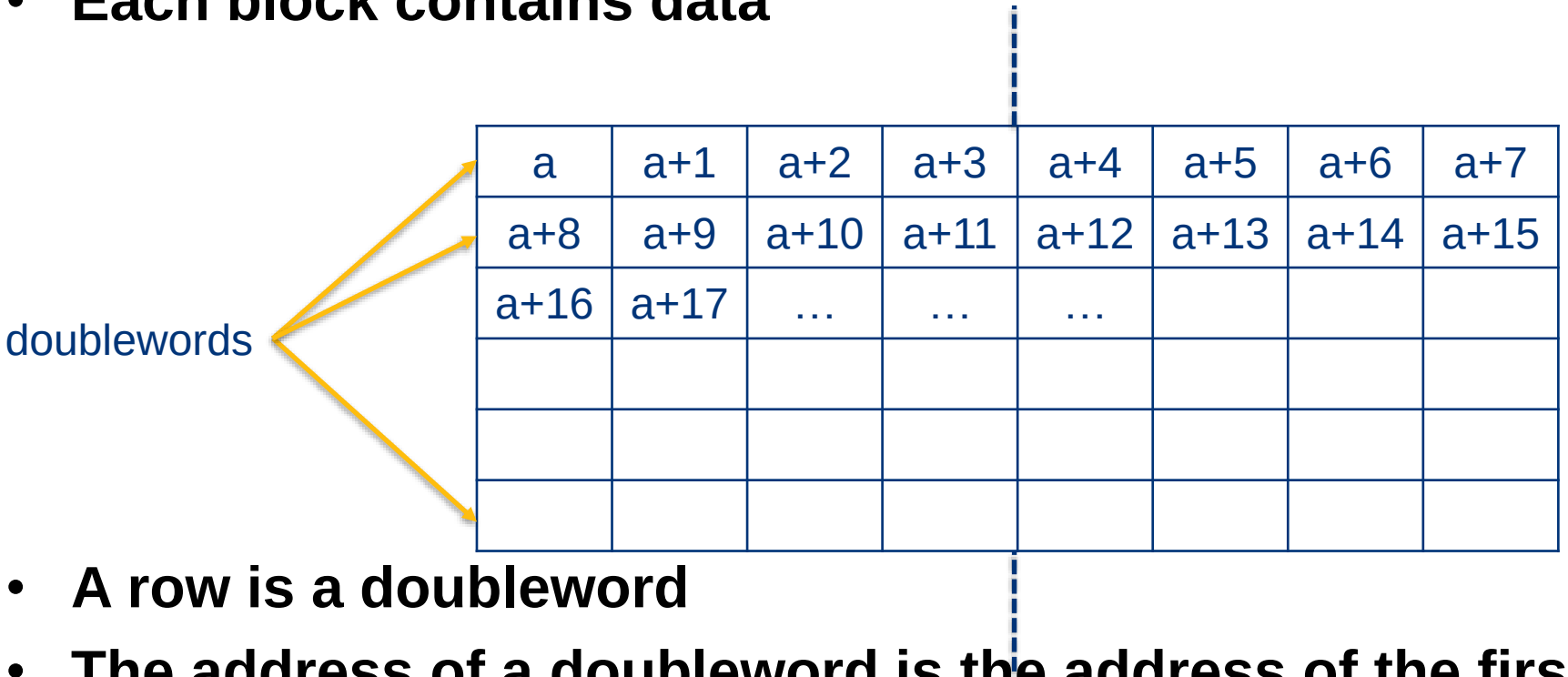
Memory Operands

- **Each block has address**
- **Each block contains data**



Memory Operands

- **Each block has address**
- **Each block contains data**



- **A row is a doubleword**
- **The address of a doubleword is the address of the first block**

C to Memory Perspective

- **We commonly define variables in C program**

int a

long int a

etc.

- **I wrote a simple code to check how much these variable declaration reserves in memory**

```
 1 #include <limits.h>
 2 #include <stdio.h>
 3
 4 int main(void) {
 5     printf("short is %d bits\n",     CHAR_BIT * sizeof( short )    );
 6     printf("int is %d bits\n",       CHAR_BIT * sizeof( int  )     );
 7     printf("long is %d bits\n",      CHAR_BIT * sizeof( long )     );
 8     printf("long long is %d bits\n", CHAR_BIT * sizeof(long long) );
 9     return 0;
10 }
```

C to Memory Perspective

- **We commonly define variables in C program**

int a

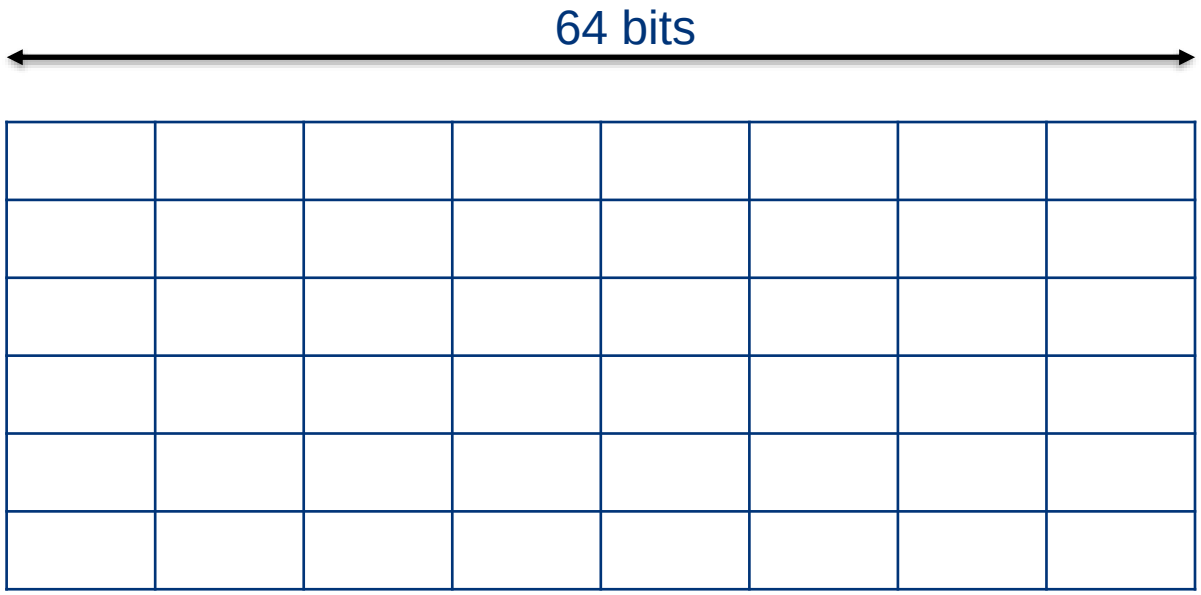
long int a

etc.

- **I wrote a simple code to check how much these variable declaration reserves in memory**

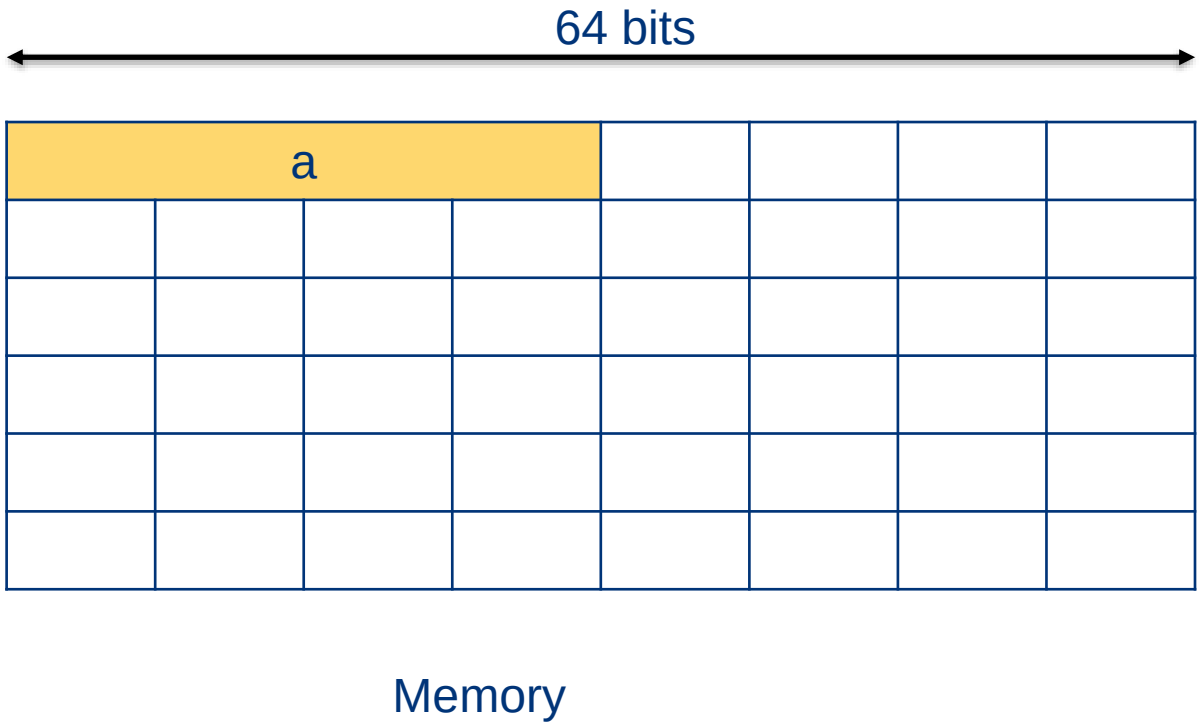
```
[COE-ECE-ad3639:c-code ad3639$ ./a.out
 short is 16 bits
 int is 32 bits
 long is 64 bits
 long long is 64 bits
[COE-ECE-ad3639:c-code ad3639$ vi try.c
 COE-ECE-ad3639:c-code ad3639$
```

C to Memory Perspective



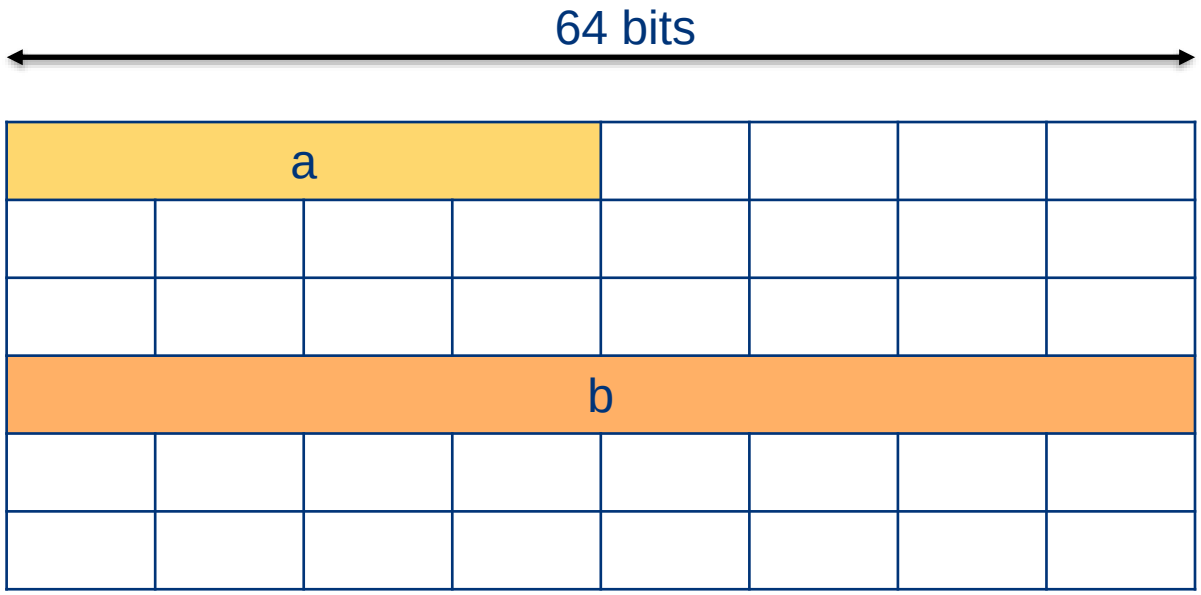
C to Memory Perspective

- **int a**



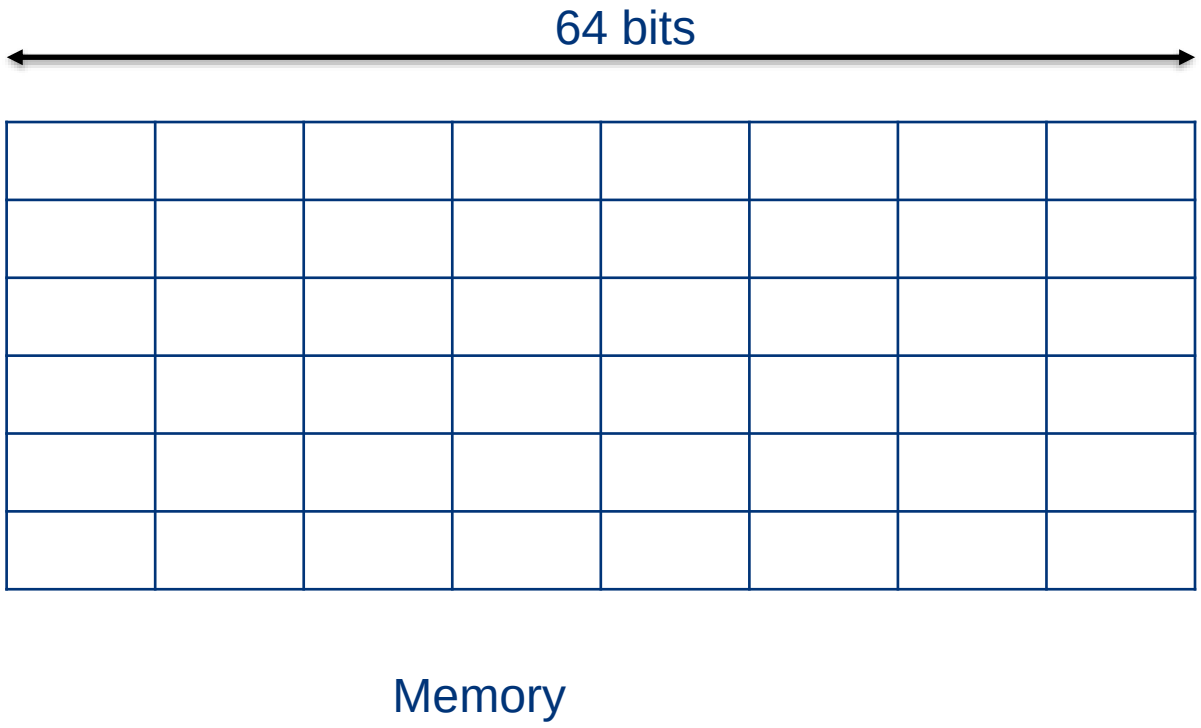
C to Memory Perspective

- **int a**
- **long int b**



C to Memory Perspective

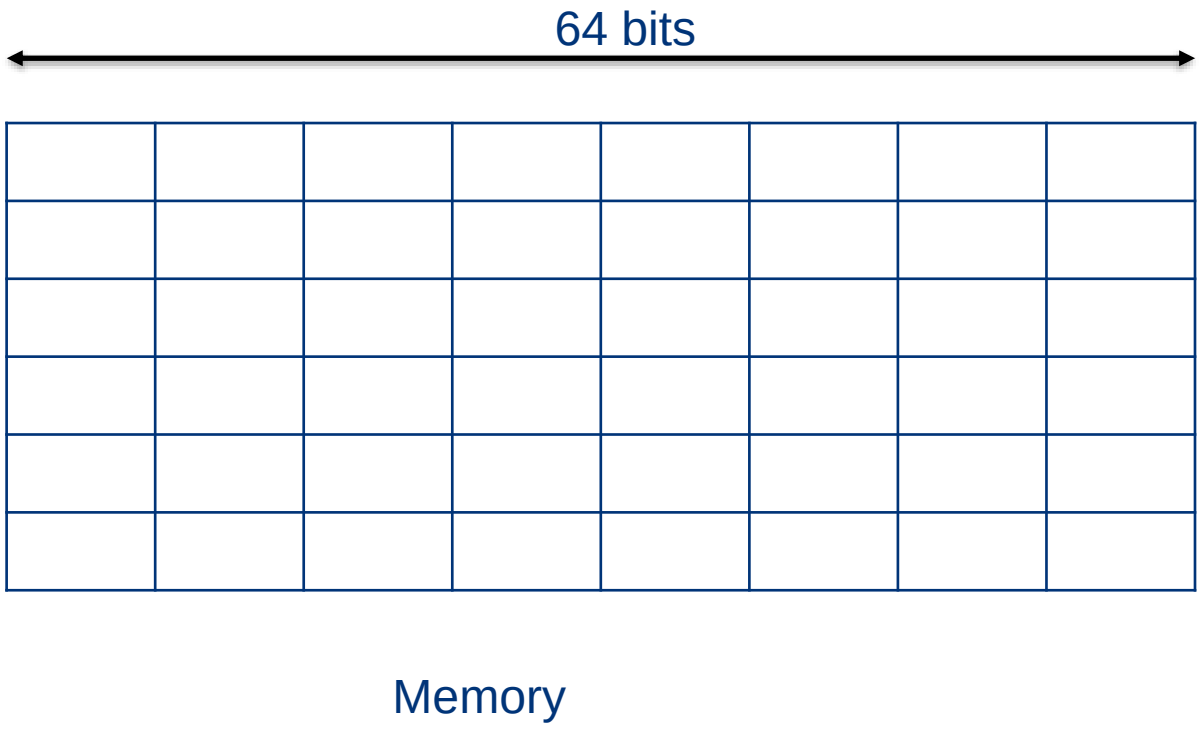
- **What about arrays?**



C to Memory Perspective

- **What about arrays?**
- **Lets say we define**

  int data[16];

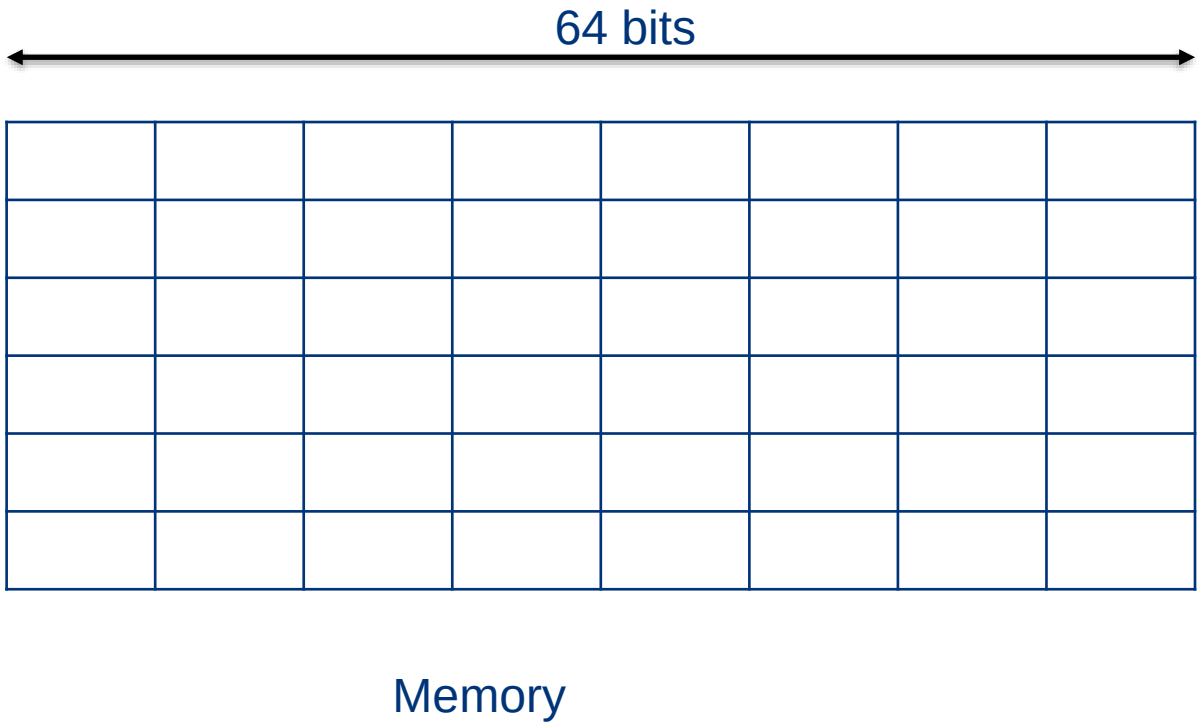


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array

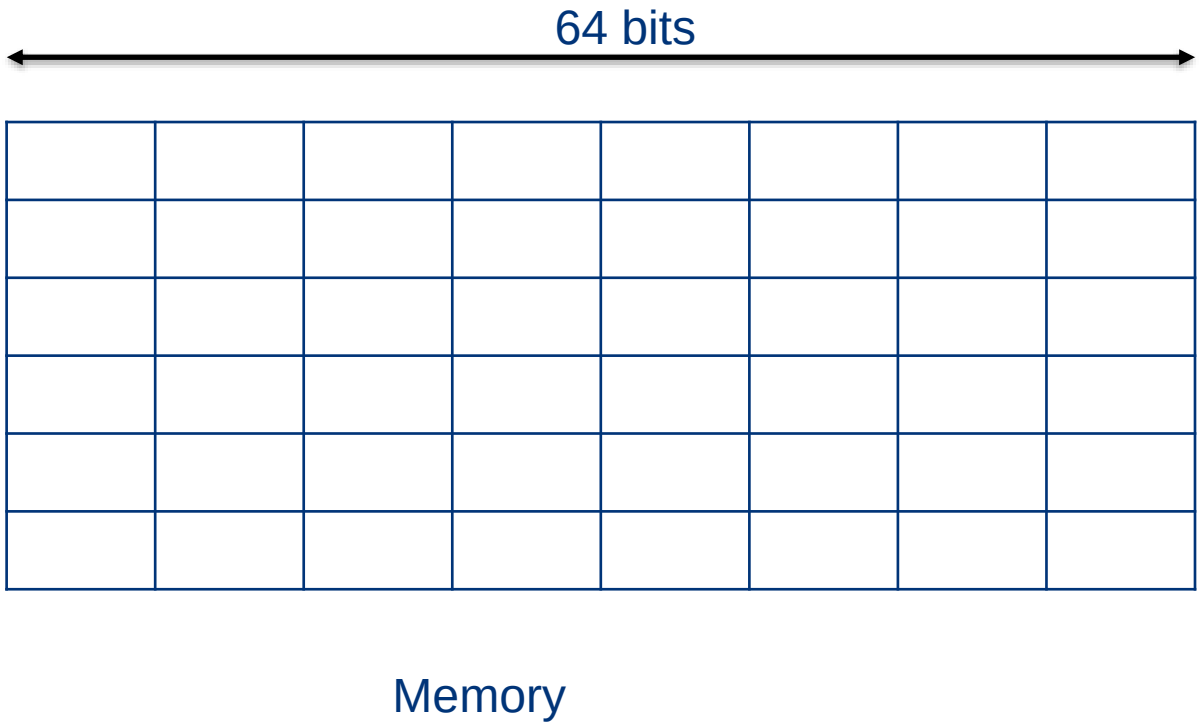


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int

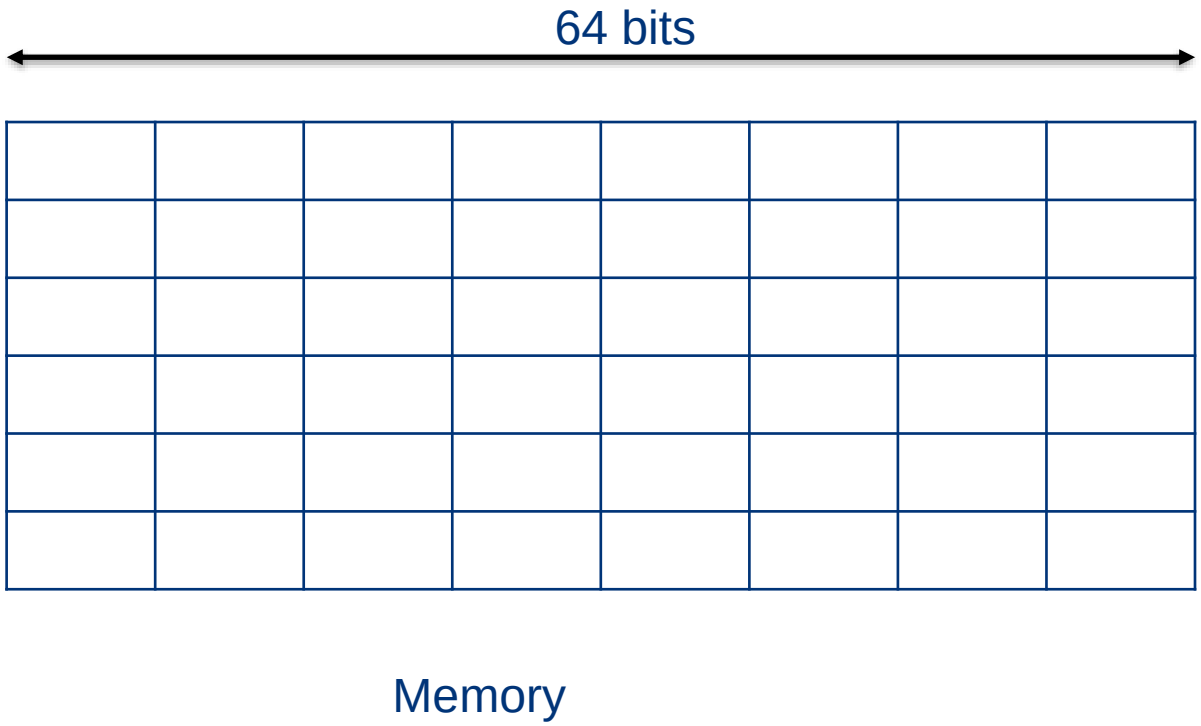


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int
- There are 16 elements in the array

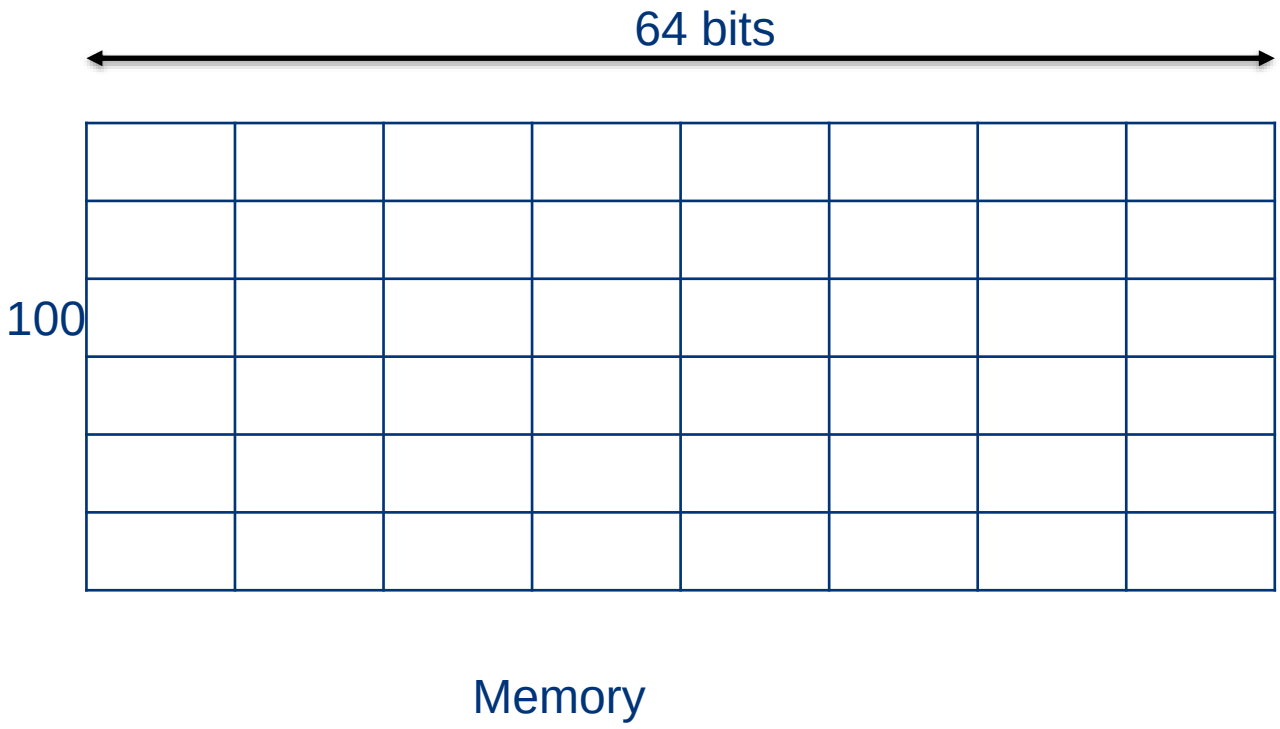


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int
- There are 16 elements in the array

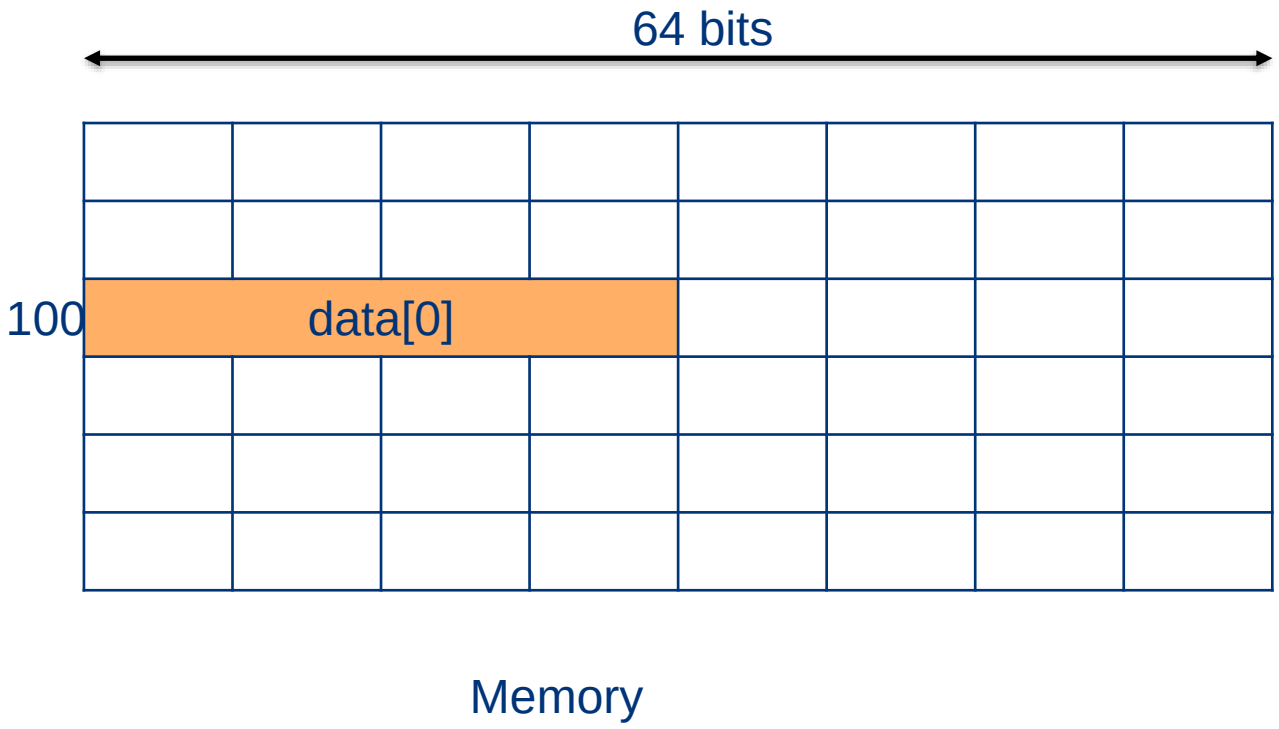


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int
- There are 16 elements in the array

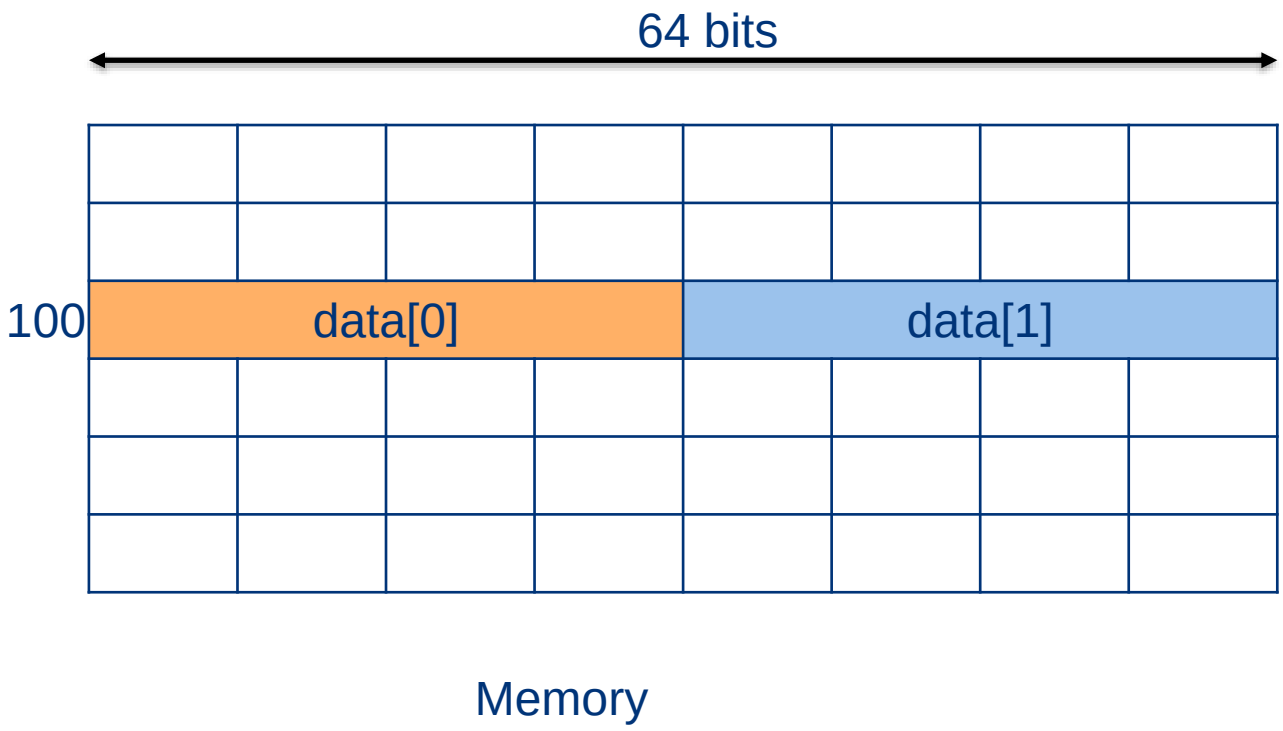


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int
- There are 16 elements in the array

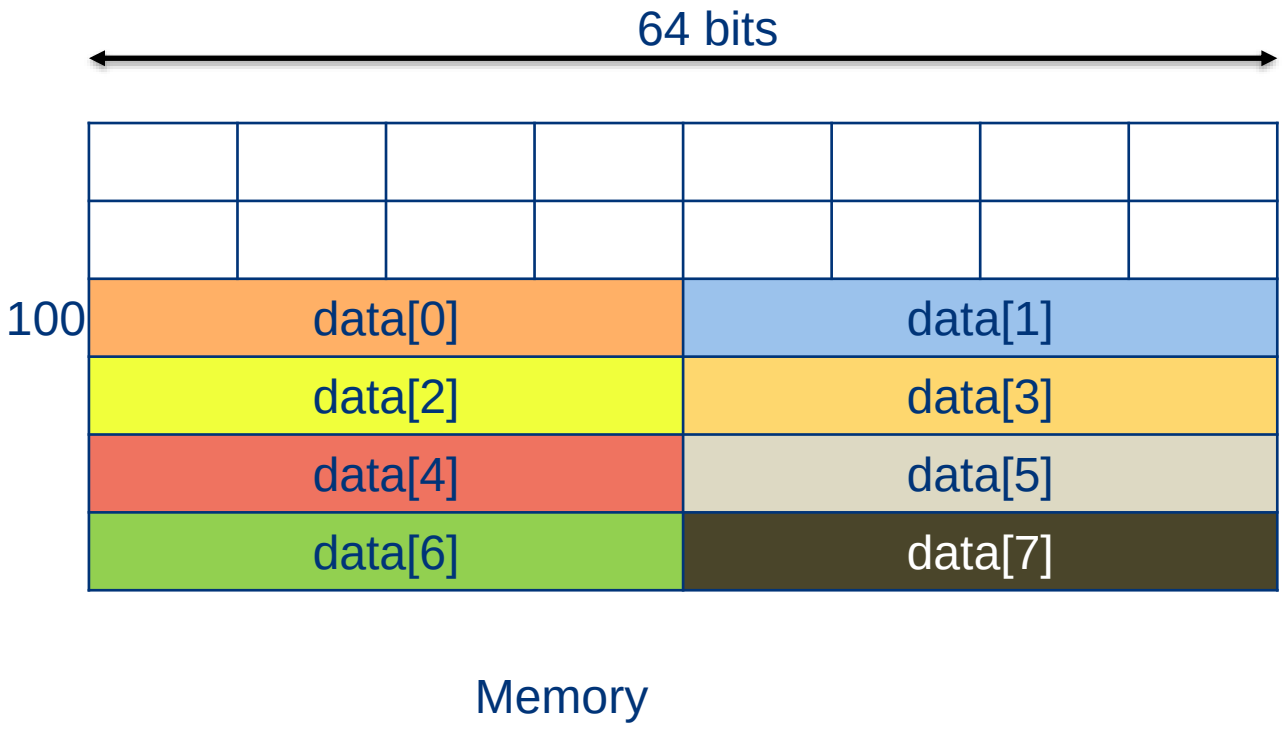


C to Memory Perspective

- What about arrays?
- Lets say we define

  int data[16];

- data is an array
- Each element is of type int
- There are 16 elements in the array

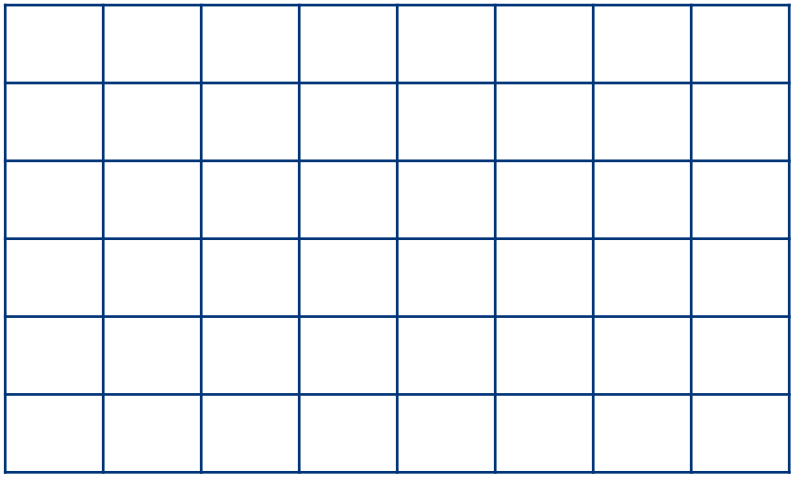


C to Memory Perspective

- **What is the address offset for each element of the array int data[16] ?**
  - 4
- **What is the address offset for each element of the array short data[16] ?**
  - 2
- **What is the address offset for each element of the array long data[16] ?**
  - 8
- **RISC-V is byte addressable ISA (important)**
  - Every block (i.e., byte) can be uniquely addressed

Memory Operands

- **Each block has address**
- **Each block contains data**



Memory Operands

- Each block has address
- Each block contains data

- Lets look at data placement
- 0001000001000001001011111100000000000…000
- How do you place this 64 bit data in a memory doubleword?

Memory Operands

- Each block has address
- Each block contains data

00010000010000010010111111000000000000…000

Memory Operands

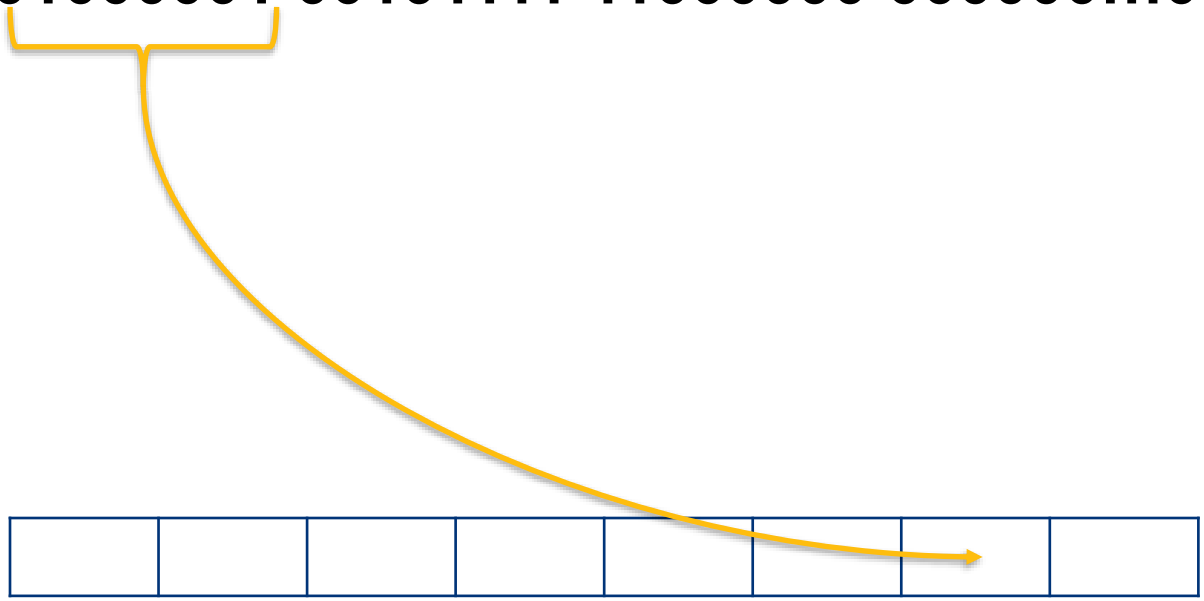
- Each block has address
- Each block contains data

00010000 01000001 00101111 11000000 000000…000

Memory Operands

- Each block has address
- Each block contains data

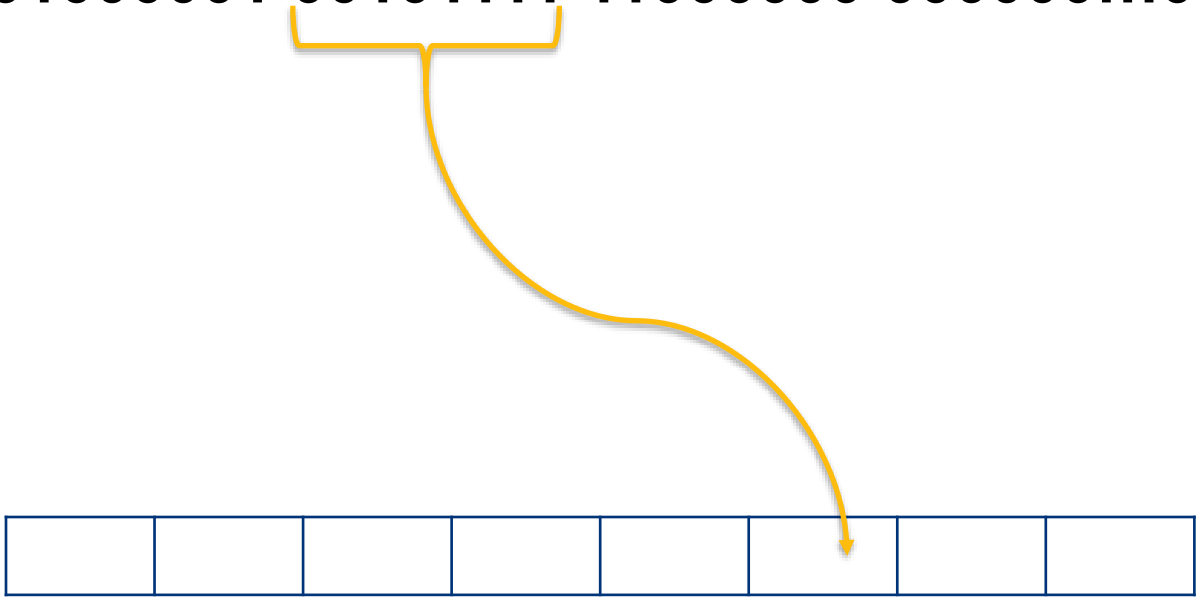
00010000 01000001 00101111 11000000 000000…000



Memory Operands

- Each block has address
- Each block contains data

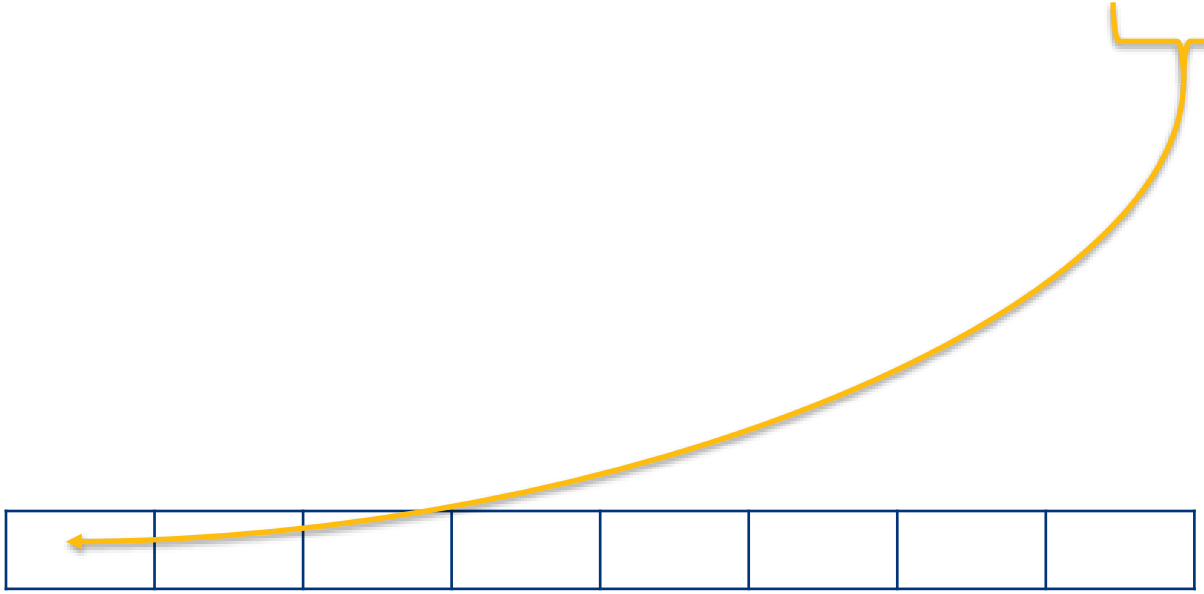
00010000 01000001 00101111 11000000 000000…000



Memory Operands

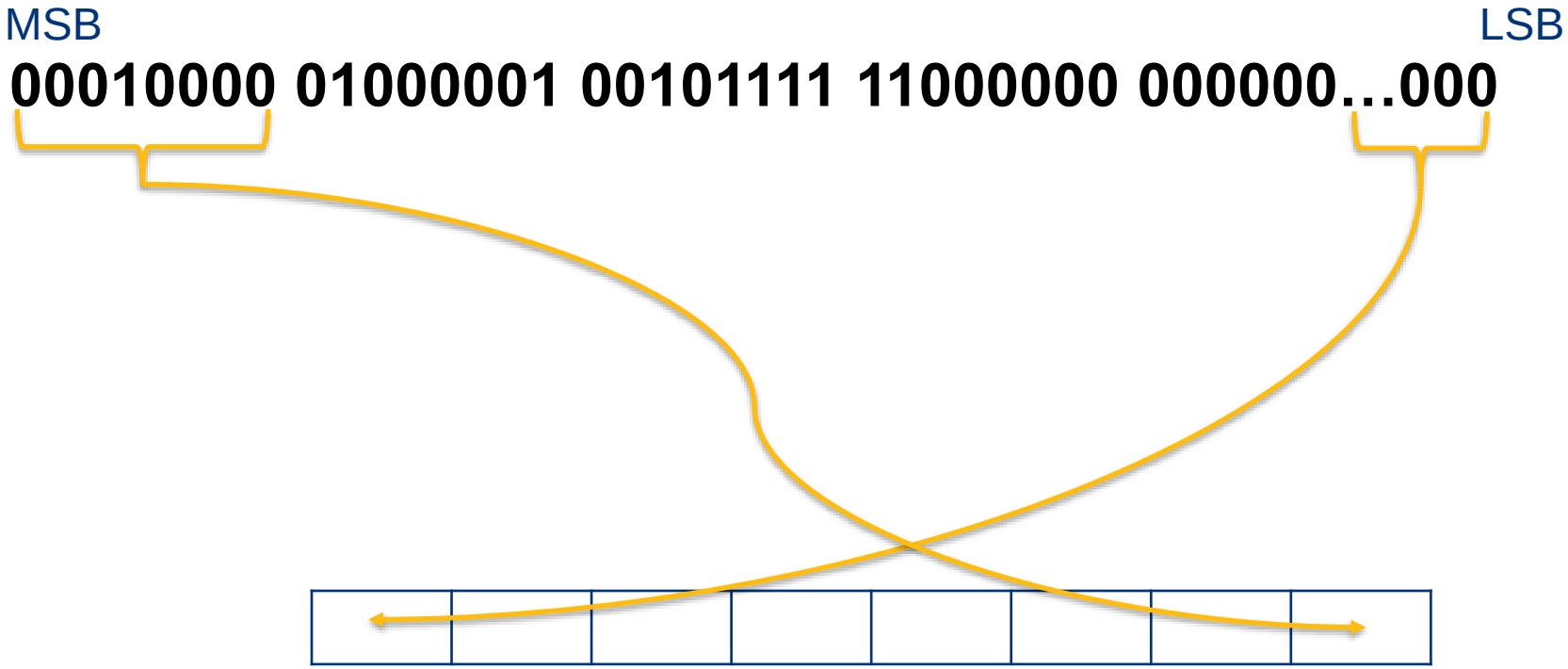
- Each block has address
- Each block contains data

00010000 01000001 00101111 11000000 000000…000



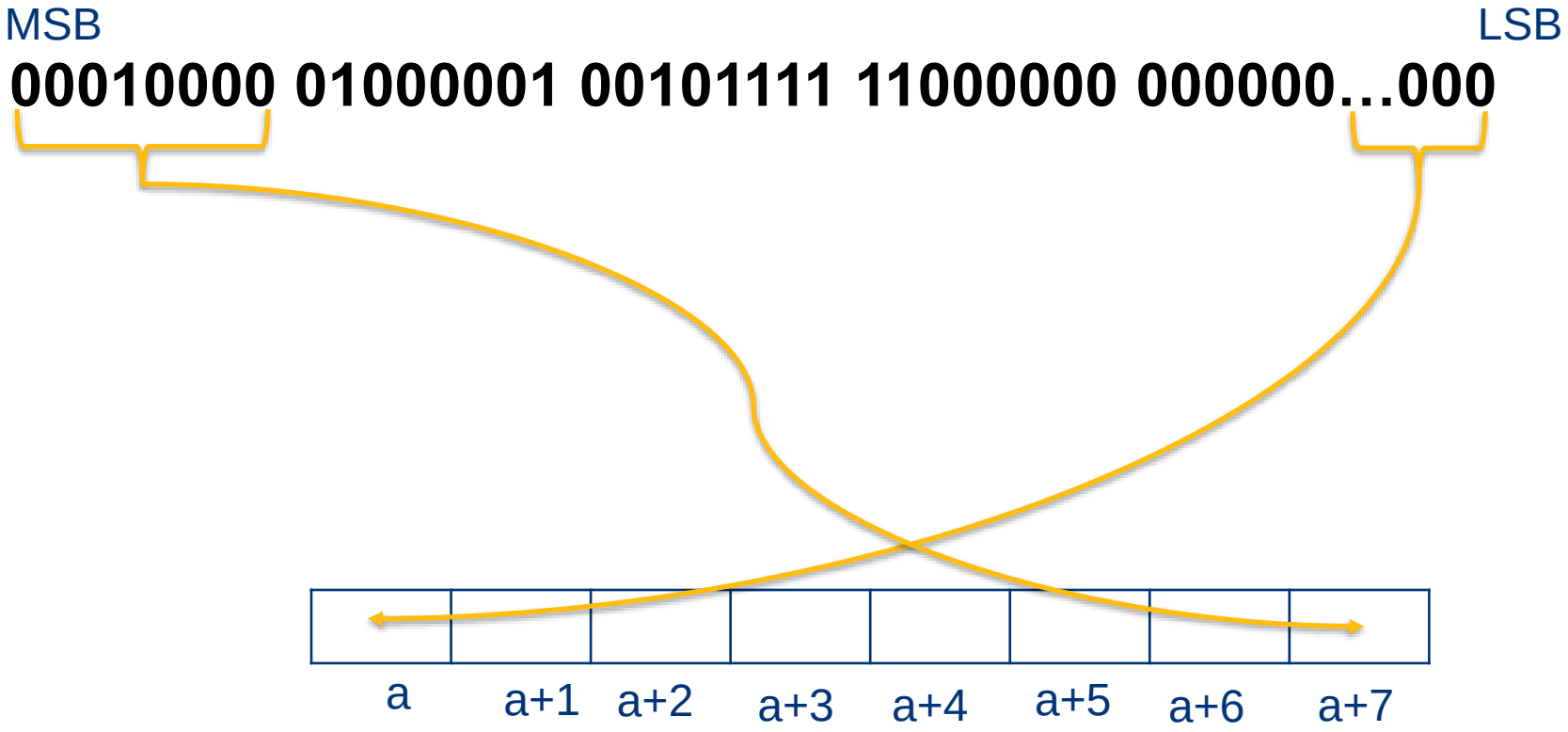
Memory Operands

- Each block has address
- Each block contains data



Memory Operands

- **Each block has address**
- **Each block contains data**



Memory Operands

- **Each block has address**
- **Each block contains data**

- **RISC-V is Little Endian (important)**
  - Least-significant byte at least address of a doubleword

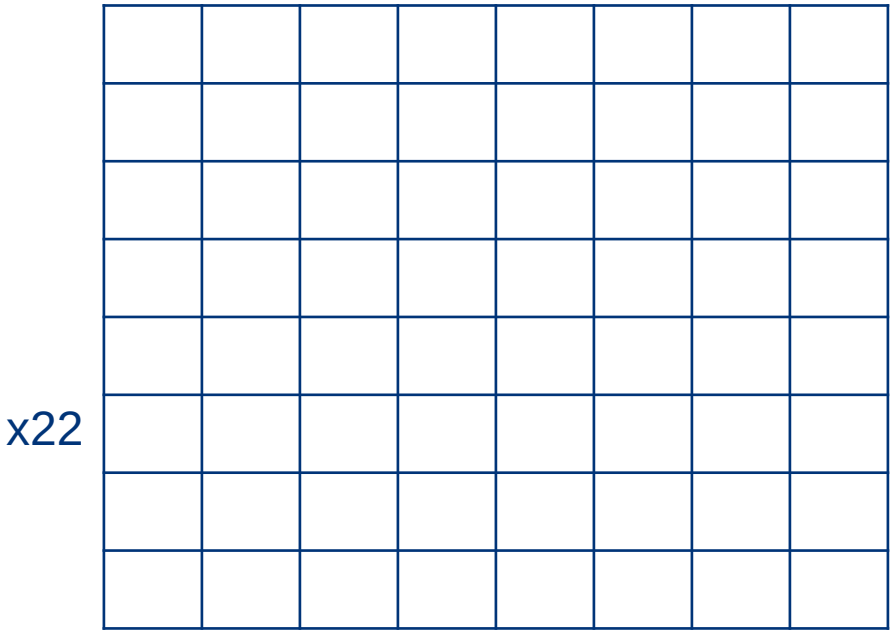
Memory Operand Example

- **C code:**

  long int A[32];

  `A[12] = h + A[8];`

  – Let the variable h in x21, base address of A in x22



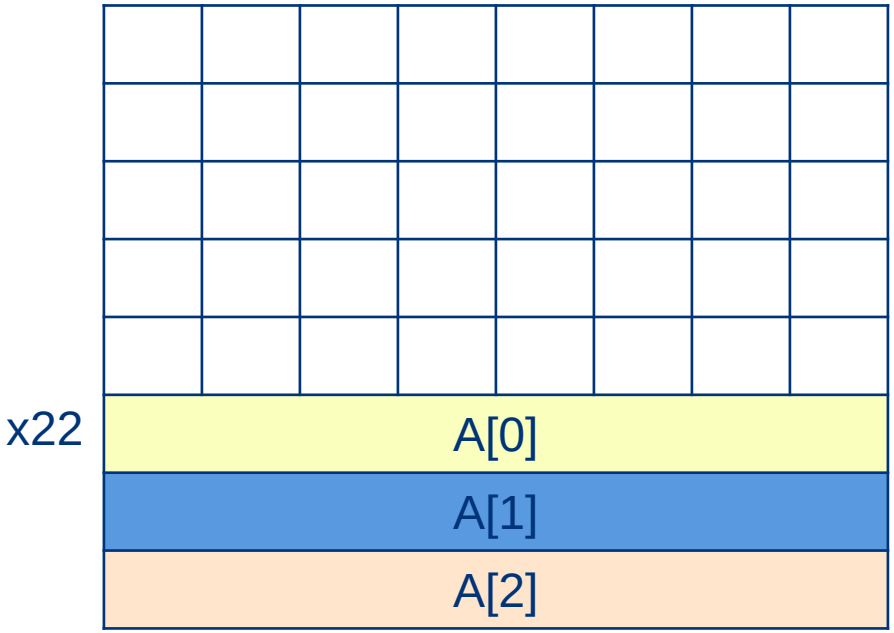
Memory Operand Example

- **C code:**

  long int A[32];

  `A[12] = h + A[8];`

  – Let the variable h in x21, base address of A in x22



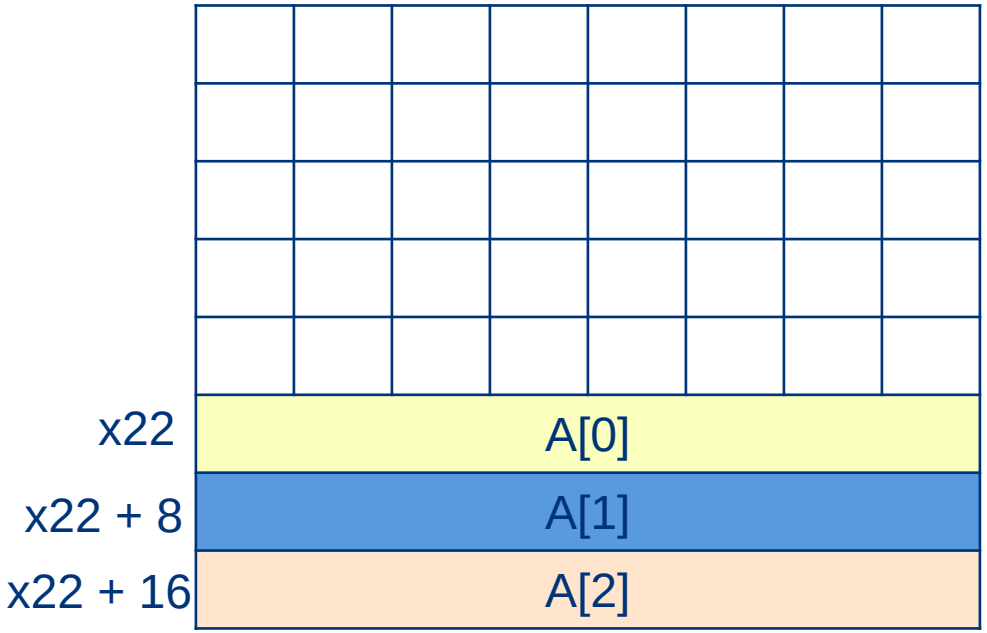
Memory Operand Example

- **C code:**

  long int A[32];

  `A[12] = h + A[8];`

  – Let the variable h in x21, base address of A in x22



Memory Operand Example

- **C code:**

  long int A[32];

  ```
  A[12] = h + A[8];
  ```

  – Let the variable h in x21, base address of A in x22

- **Compiled RISC-V code:**
  – Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld    x9, 64(x22)      //load A[8] in x9
add   x9, x21, x9      // x9 = h + x9
sd    x9, 96(x22)      // store x9 as A[12]
```

Memory Operations in RISC-V

- **ld = load double word (Memory to CPU)**
- **sd = store double word (CPU to Memory)**

- **lb = load byte (RISC-V is btte addressable)**
- **lh = load half word (i.e., load 2 bytes)**
- **lw = load word (i.e., load 1 word, i.e., 32 bits)**
- **Same instructions for store**
  - sb, sh, and sw
- **Question: Does RISC-V support aligned or unaligned memory transfers?**

RISC-V Operations

- Register Operations
- Memory Operations
- **Immediate Operations**

Immediate Operands

- **Constant data specified in an instruction**

```
addi x22, x22, 4        // x22 = x22 + 4
```

```
Remember x22 is 64 bits
So, decimal 4 needs to be represented in 64 bits
```

Unsigned Binary Integers

- **Given an n-bit number**
- **What is the range of numbers that can be represented**
- **Binary to decimal conversion**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **Range: 0 to $+2^n - 1$**
- **Example**
  - $0000\ 0000 \ldots 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- **Using 64 bits: 0 to +18,446,774,073,709,551,615**

2s-Complement Signed Integers

- **Given an n-bit number**
- **Range: -2^{n-1} to $+2^{n-1} - 1$**
- **Example**
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- **Using 64 bits: −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**

2s-Complement Signed Integers

- **Bit 63 is sign bit**
  - 1 for negative numbers
  - 0 for non-negative numbers
- **Non-negative numbers have the same unsigned and 2s-complement representation**
- **Some specific numbers**
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- **Complement and add 1**
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- **Example: negate +2**
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

Sign Extension

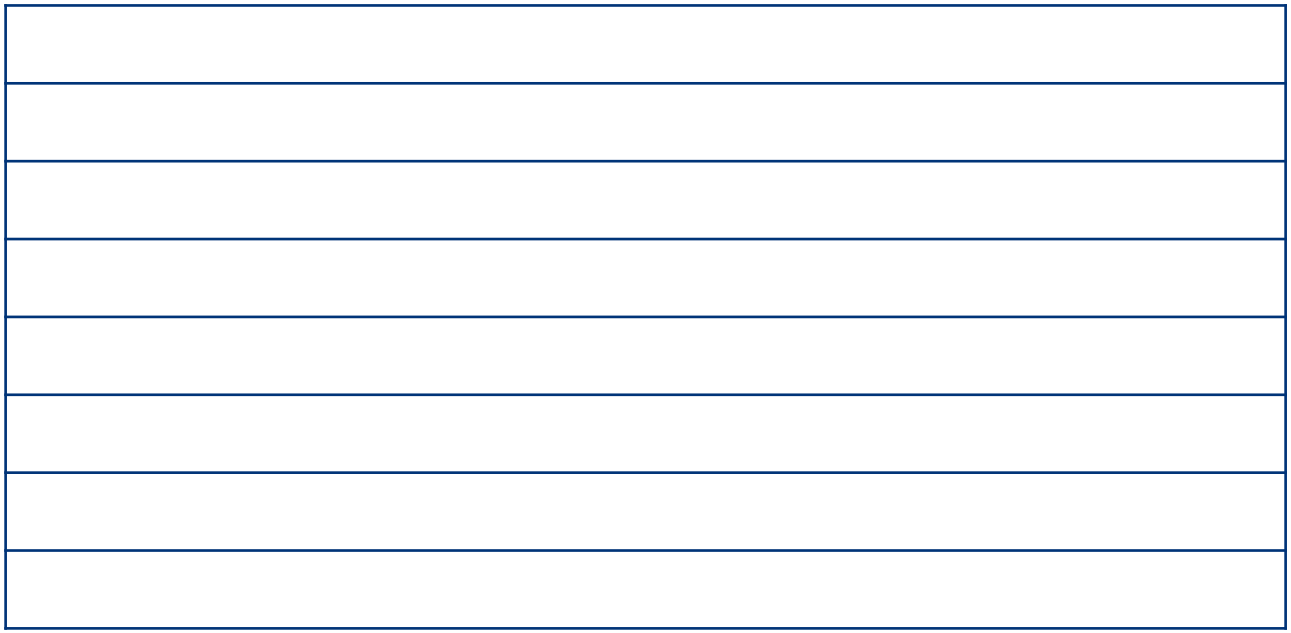
- **Representing a number using more bits**
  - Preserve the numeric value
- **Replicate the sign bit to the left**
- **Examples: 8-bit to 16-bit**
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- **In RISC-V instruction set**
  - lb: sign-extend loaded byte
  - lbu: zero-extend loaded byte

Lecture thus far (recap)

- **3 types of instructions**
  - Register instructions
  - Memory instructions
  - Immediate instructions

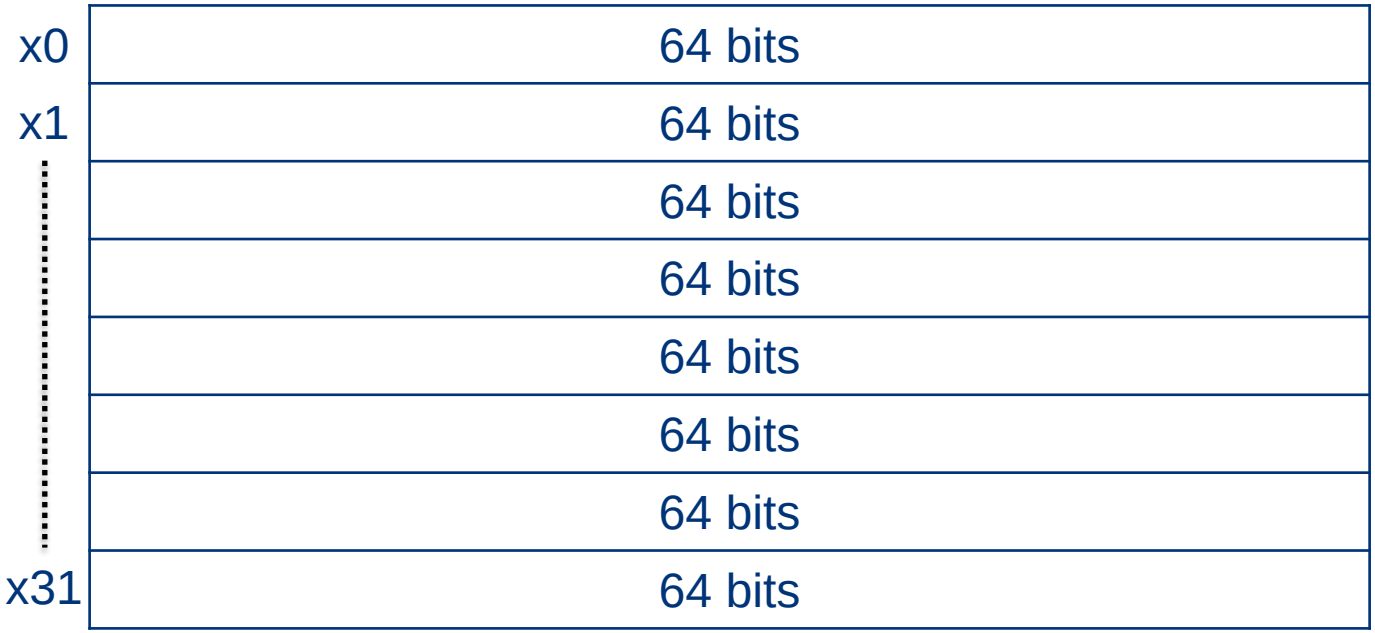
Register Files

- **32 registers of 64-bit each**



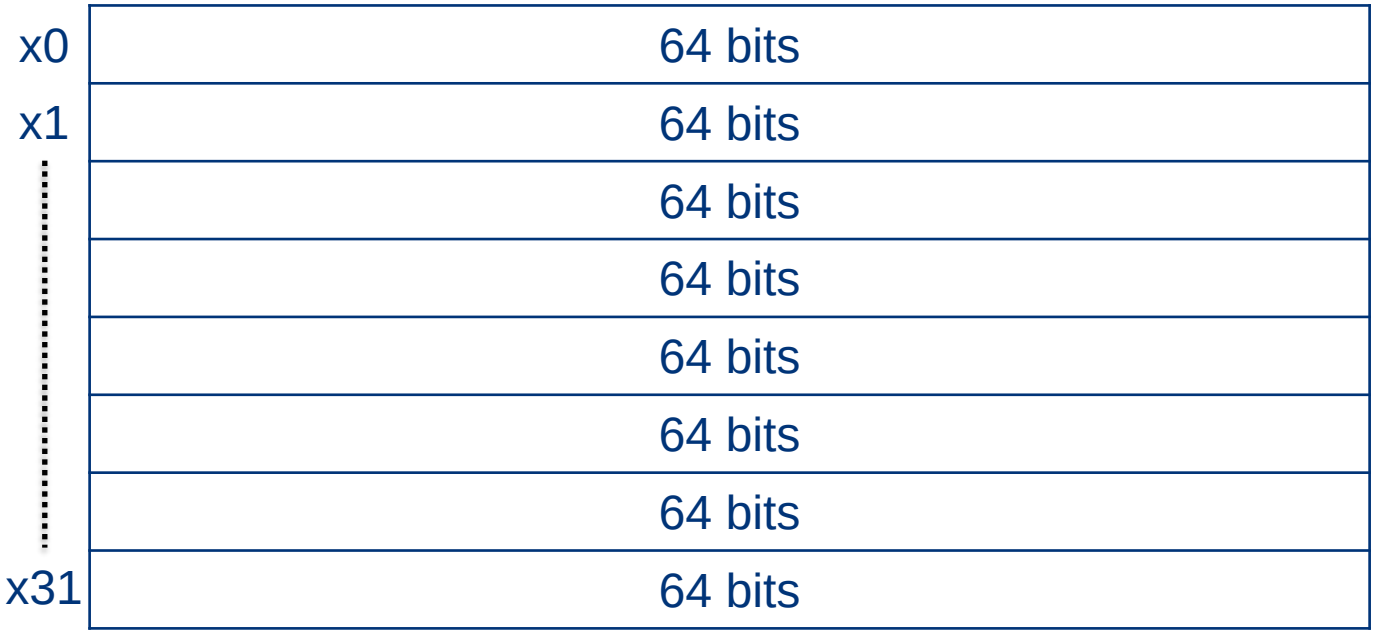
Register Files

- **32 registers of 64-bit each**



Register Files

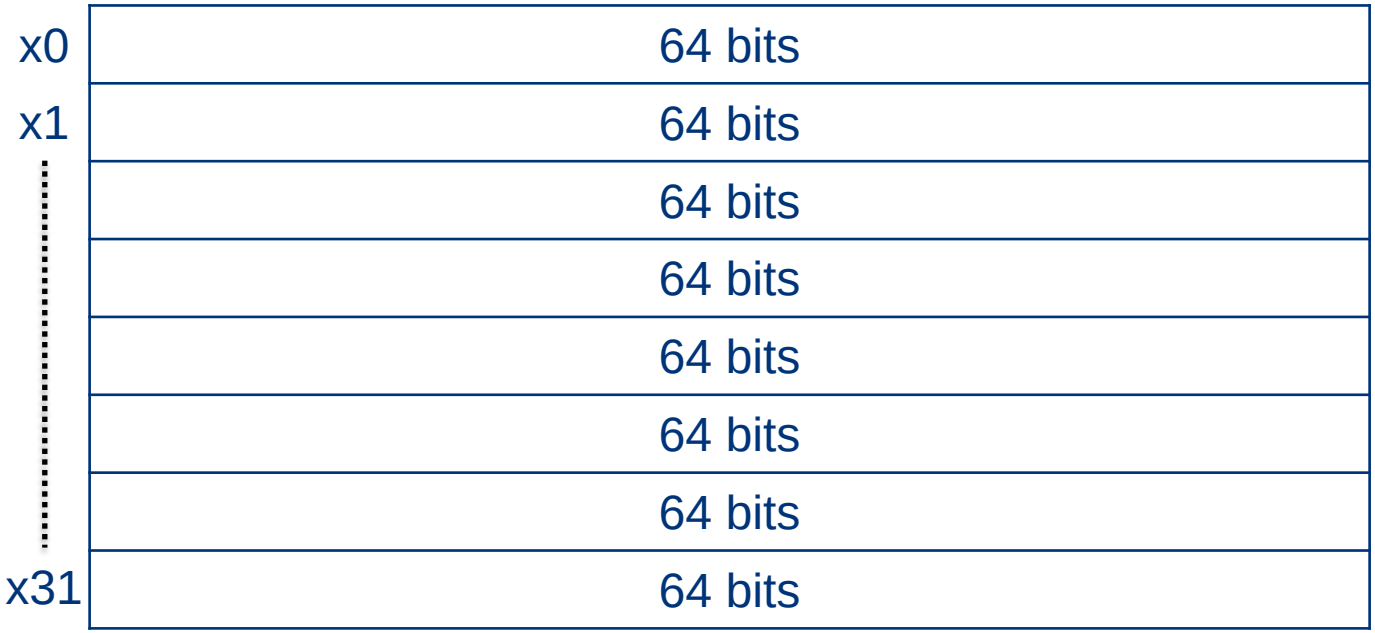
- **32 registers of 64-bit each**



Question: How many bits are needed to represent the address of the register file?

Register Files

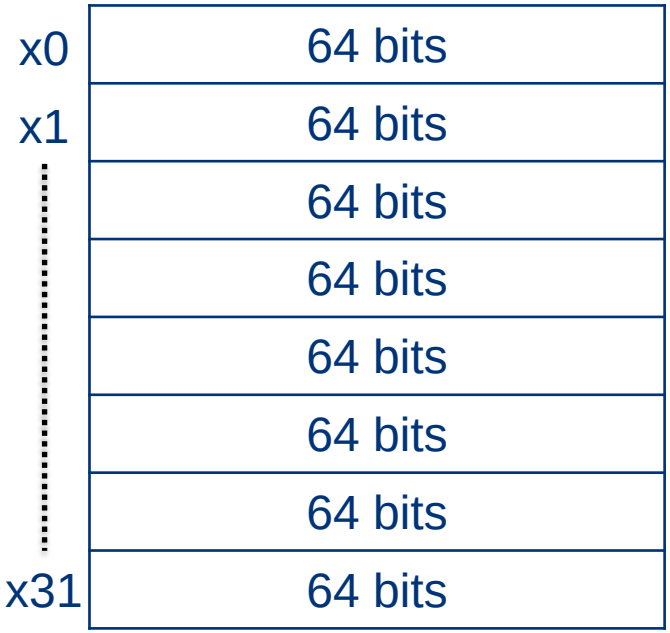
- **32 registers of 64-bit each**



Question: How many bits are needed to represent the address of the register file?
And: log2(32) = 5 bits

Register Files

- **32 registers of 64-bit each**
- **00000 ➔ x0**
- **00001 ➔ x1**
- **00010 ➔ x2**
- **…**
- **11111 ➔ x31**



Question: How many bits are needed to represent the address of the register file?
And: log2(32) = 5 bits

Representing Instructions

- **Instructions are encoded in binary**
  - Called machine code
  - RISC-V: all instructions are 32 bits

- **RISC-V instructions**
  - `ld  x9, 64(x22) // we have seen this instruction before`
    - How to represent it in binary?
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V R-format Instructions

- **Register format of instructions**
- **Instruction fields**
  - opcode: operation code (add, sub, etc)
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example


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

R-format Example


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

funct7 = funct3 = 0

R-format Example


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

R-format Example


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

add = 0110011 (green page of the P&H book)

R-format Example


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

add = 0110011 (green page of the P&H book)

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = ?

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

rs2 = x21 ➔ 10101

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

rs2 = x21 ➔ 10101

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

rs2 = x21 ➔ 10101

rd = x9 ➔ 01001

R-format Example


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

add = 0110011 (green page of the P&H book)

rs1 = x20 ➔ 10100

rs2 = x21 ➔ 10101

rd = x9 ➔ 01001

R-format Example


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

00000001010110100000010010110011

R-Format Example

- **Also used for all logical operations**
  - and x9, x8, x7 // x9 = x8 & x7 (bitwise and)
  - and = 0110011
  - funct7 = 00000 and funct3 = 111
  - Find the 32-bit instructions

  - funct3 and funct7 are specified in green booklet

RISC-V I-format Instructions

- **Immediate format of instruction**

- **addi x12, x17, 4**
- **addi = 0010011**
- **funct3 = 000**
- **rs1 = x17 ➔ 10001**
- **rd = x12 ➔ 01100**
- **immediate = 4 ➔ 000000000100 (in 12 bits)**

- **000000000100 10001 000 01100 0010011**

RISC-V I Format

- **Apart from immediate instructions, the I format is also used for**
  - andi (AND immediate), ori (OR immediate), etc
  - load instructions

  - `ld  x9, 64(x22) // x9 = Memory[x22 + 64]`
  - `ld = 0000011, funct3 = 011`
  - rs1 = x22 ➔ 10110
  - rd = x9 ➔ 01001
  - immediate = 64 ➔ 000001000000
  - 000001000000 10110 011 01001 0000011

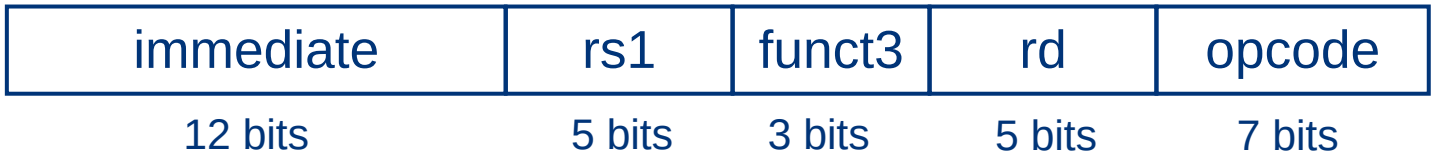
RISC-V S-format Instructions

- **Store format (storing register to memory)**

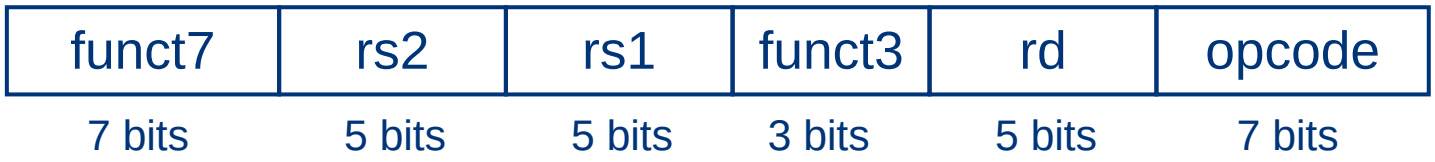
- **sd x11, 64(x17)**
- **Conceptually store is also a memory operation like load, but store needs a whole new format and cannot fit in the immediate format. Why?**
- **Memory[x17+64] = x11**
- **Observe the destination is a memory and not a register**
- **ld x5, 36(x7) // x5 = Memory[36 + x7]**
- **So, in store instruction, no destination field is needed**

RISC-V S Format

- **Cannot fit the store instruction in this format**



- **Can it fit in the R-format?**



- **sd x11, 64(x17) ➔ a number (64) needs immediate field**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 000001000000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 0000010 00000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 0000010 00000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 0000010 00000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 000001000000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

RISC-V S-format Instructions

- **Store format (storing register to memory)**

- **sd x11, 64(x17)**
- **sd = 0100011, funct3 = 011**
- **rs2 = x11 ➔ 01011**
- **rs1 = x17 ➔ 10001**
- **immediate = 64 ➔ 000001000000**
- **Observe immediate field is split (we will answer why soon)**
- **0000010 01011 10001 011 00000 0100011**

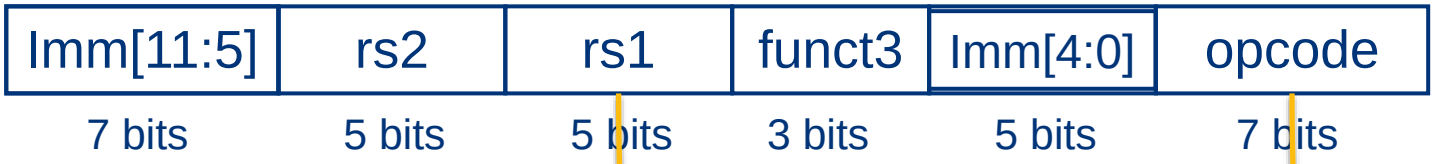
RISC-V S-format Instructions

- **Store format (storing register to memory)**

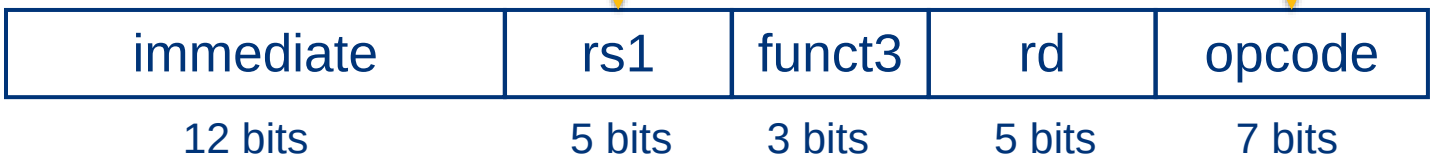
- **Different immediate format for store instructions**
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: split into 2 positions

RISC-V S-format Instructions

- **Store format (storing register to memory)**



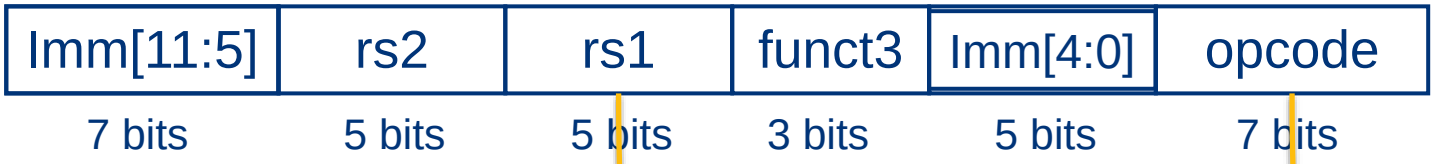
- **Immediate format**



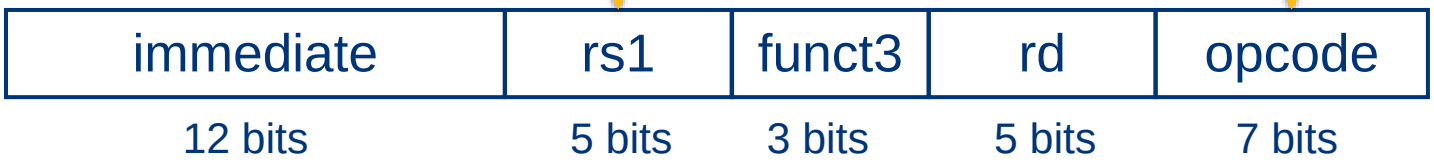
rs1 starts exactly at the same position for both format

RISC-V S-format Instructions

- **Store format (storing register to memory)**



- **Immediate format**



- **R-format**

Design Principle

- ***Design Principle 3:*** **Good design demands good compromises**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V I Format

- **Apart from immediate instructions, the I format is also used for**
  - Immediate instructions
  - Load Instructions
  - Logical instructions

Logical Operations

- **Instructions for bitwise manipulation**

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	~	~	

Shift Operations

- **immed: how many positions to shift**
- **Shift left logical**
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

RISC-V I Format

- **Apart from immediate instructions, the I format is also used for**
  - Immediate instructions
  - Load Instructions
  - Logical instructions
  - Conditional operations

Conditional Operations

- **Branch to a labeled instruction if a condition is true**
  - Otherwise, continue sequentially

- **`beq rs1, rs2, L1`**
  - if (rs1 == rs2) branch to instruction labeled L1

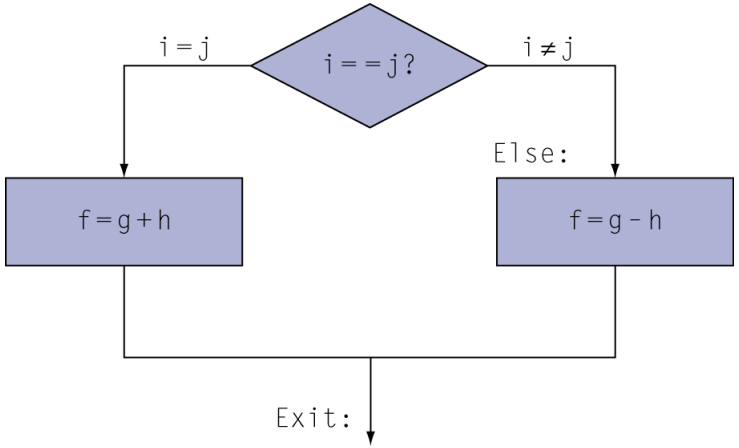
- **`bne rs1, rs2, L1`**
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

- **C code:**

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

  - f, g,h in x19, x20, x21
  - i,j in x22, x23



- **Compiled RISC-V code:**

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

Observe the use of x0

More Conditional Operations

- **`blt rs1, rs2, L1`**
  - if (rs1 < rs2) branch to instruction labeled L1
- **`bge rs1, rs2, L1`**
  - if (rs1 >= rs2) branch to instruction labeled L1
- **Example**
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- **Signed comparison: blt, bge**
- **Unsigned comparison: bltu, bgeu**
- **Example**
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1

Procedure Calling

- **Steps required**
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Procedure Call Instructions

- **Procedure call: jump and link**

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address
- **Procedure return: jump and link register**

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Procedure Types

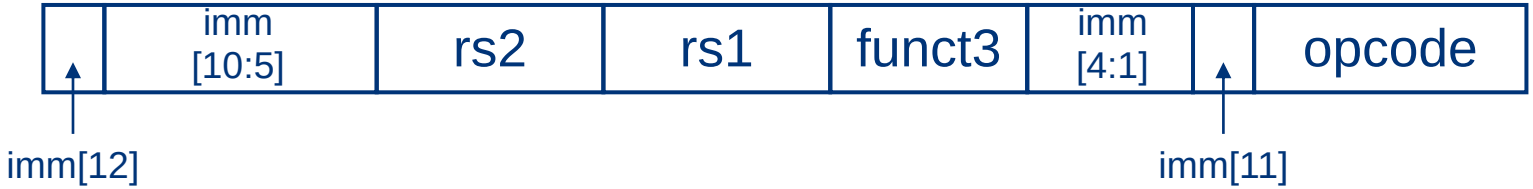
- **Leaf procedures**
  - Procedures that do not call any other procedures
    - i.e., they execute and return to the caller
- **Non-Leaf procedures**
  - Procedures that call other procedures
- **Please follow the examples on page 100**

RISC-V I Format

- **Apart from immediate instructions, the I format is also used for**
  - Memory store instruction
  - Logical instructions
  - Conditional operations
  - Branch operations

Branch Addressing

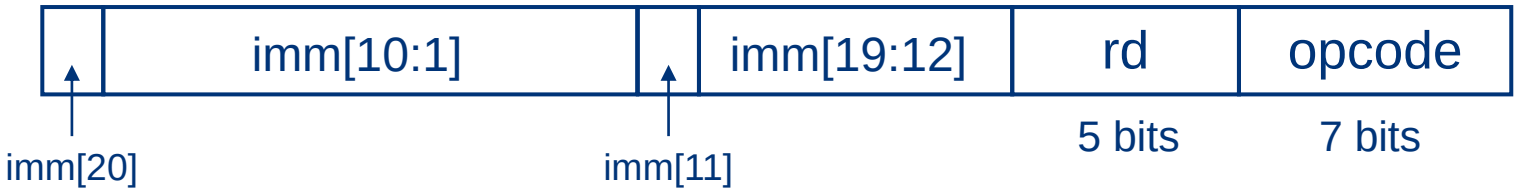
- **Branch instructions specify**
  - Opcode, two registers, target address
- **Most branch targets are near branch**
  - Forward or backward
- **Format:**



- **PC-relative addressing**
  - Target address = PC + immediate × 2

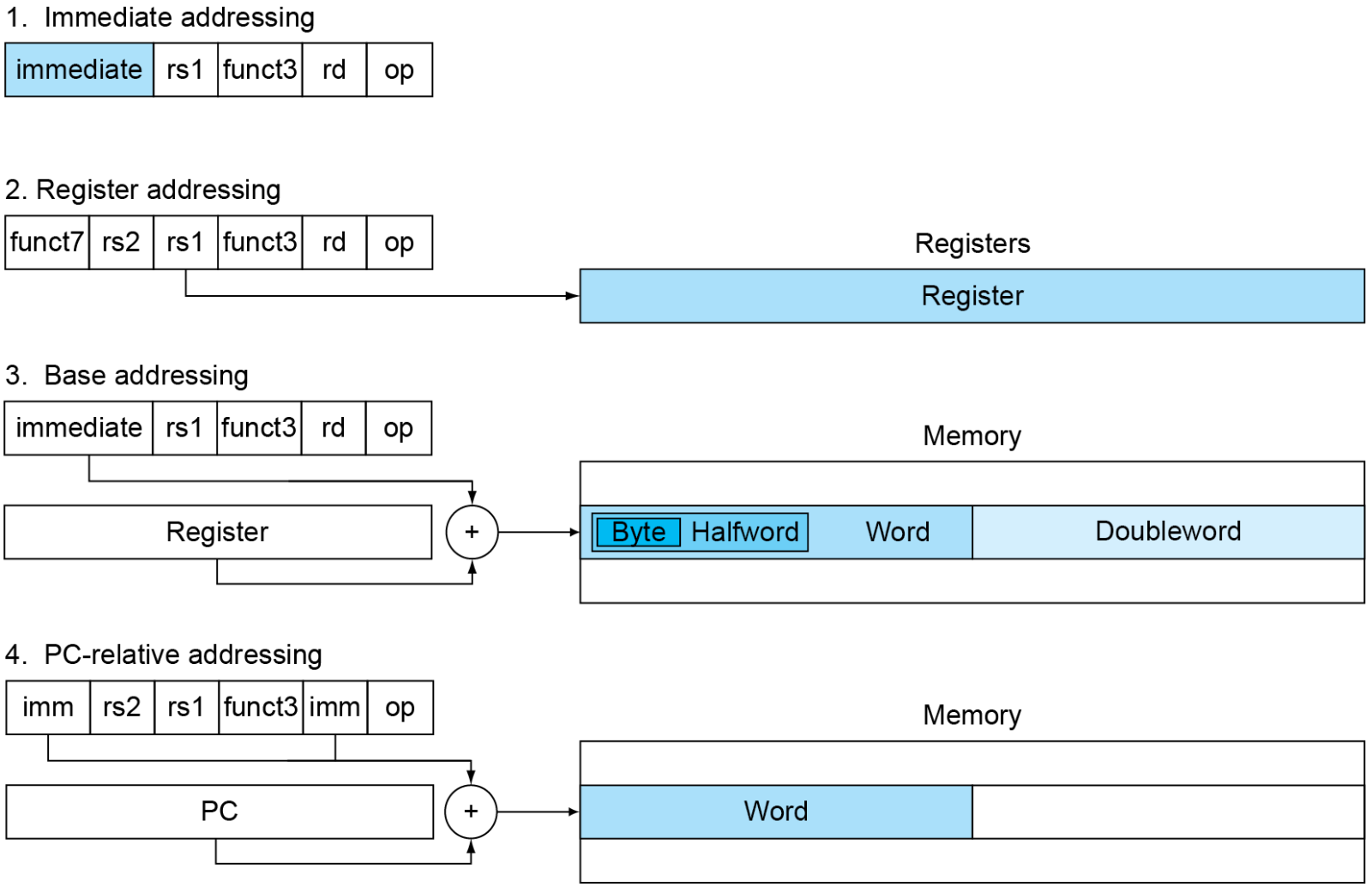
Jump Addressing

- **Jump and link (`jal`) target uses 20-bit immediate for larger range**
- **UJ format:**



- **For long jumps, eg, to 32-bit absolute address**
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary



RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Other ISA

- **x86**
- **PDP (programmed data processor)**
- **IBM 360**
- **SIMD ISA: CRAY-1**
- **VLIW ISA: IA-64**
- **PowerPC, POWER**
- **RISA ISA: Alpha, SPARC, MIPS, ARM etc**
- **What are the fundamental differences?**
  - How instructions are specified?
  - What instructions do?

ISA Semantic Gap

- **ISAs differ**
  - How complex are the instructions?

ISA complexity trade-off

- **Large semantic gap = ISA closer to the control signals**
  - Advantages
    - Simple hardware
  - Disadvantages
    - Complex compiler (translate a high level language to control signals)

- **Small sematic gaps = ISA closer to the high level language**
  - Advantages
    - Simple compiler
  - Disadvantages
    - Complex hardware

– Java machines, LISP machines, object-oriented machines, capability-based machines are all examples of small semantic gaps

Semantic Gap

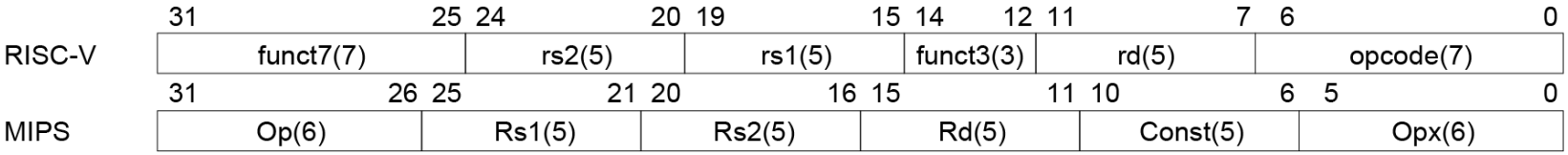
- **RISC (Reduced instruction set computer)**
  - Simple instructions
- **CISC (Complex instruction set computer)**
  - Complex instructions

MIPS ISA

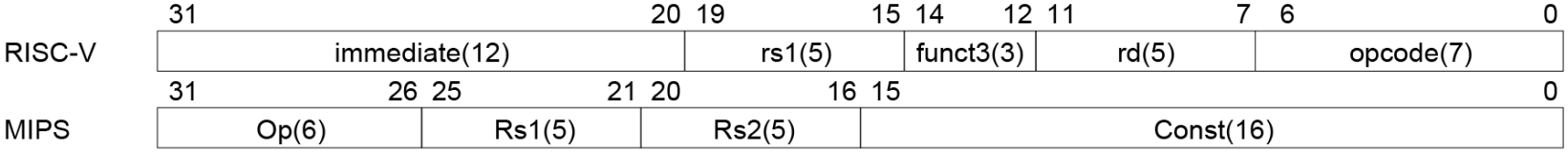
- **MIPS: commercial predecessor to RISC-V**
- **Similar basic set of instructions**
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- **Different conditional branches**
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

Instruction Encoding

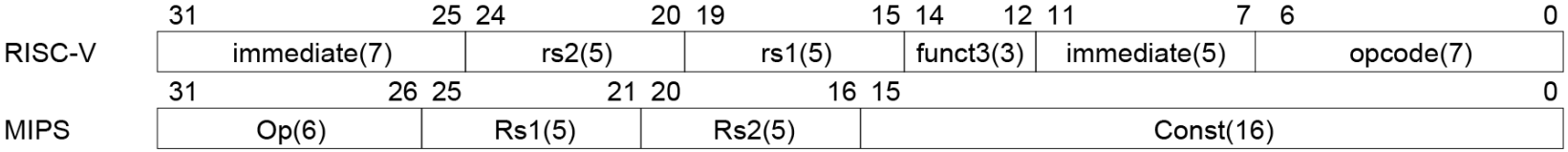
Register-register



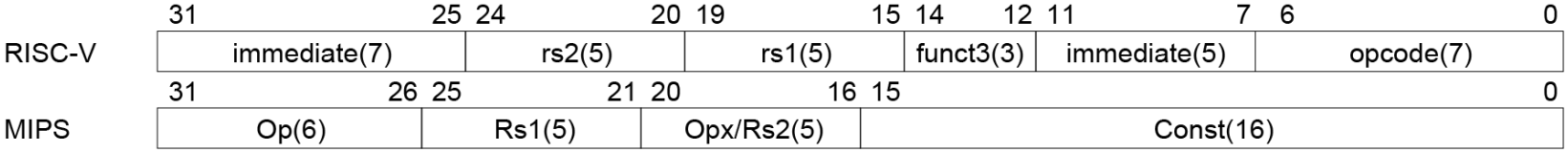
Load



Store



Branch



The Intel x86 ISA

- **Evolution with backward compatibility**
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA

- **Further evolution…**
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- **And further…**
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions

Concluding Remarks

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- **Make the common case fast**
- **Layers of software/hardware**
  - Compiler, assembler, hardware
- **RISC-V: typical of RISC ISAs**