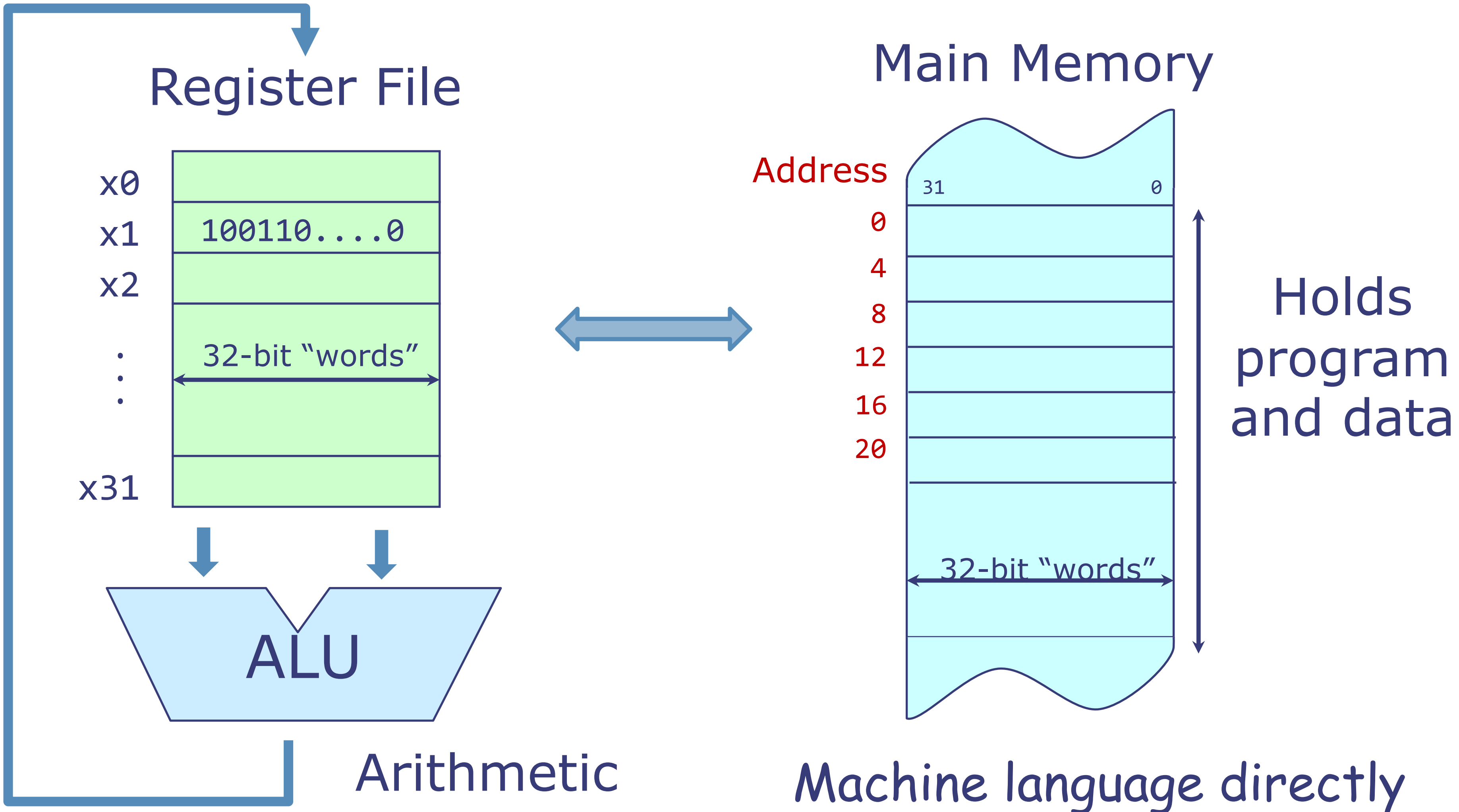
RISC-V Assembly and Binary Notation

Course Mechanics Reminders

- Course website: http://6004.mit.edu
  - All lectures, videos, tutorials, and exam material can be found under Information/Resources tab.
  - Lab handouts will be under the Labs tab.
- We use piazza extensively: piazza.com/mit/spring2019/6004
  - Post publicly when possible (can be anonymous).
  - Search piazza before posting new question.
  - For private matters, post privately.
- Tutorial problems reviewed in recitation complement lab work in assignments. Both are critical to succeed in the course.

Components of a MicroProcessor



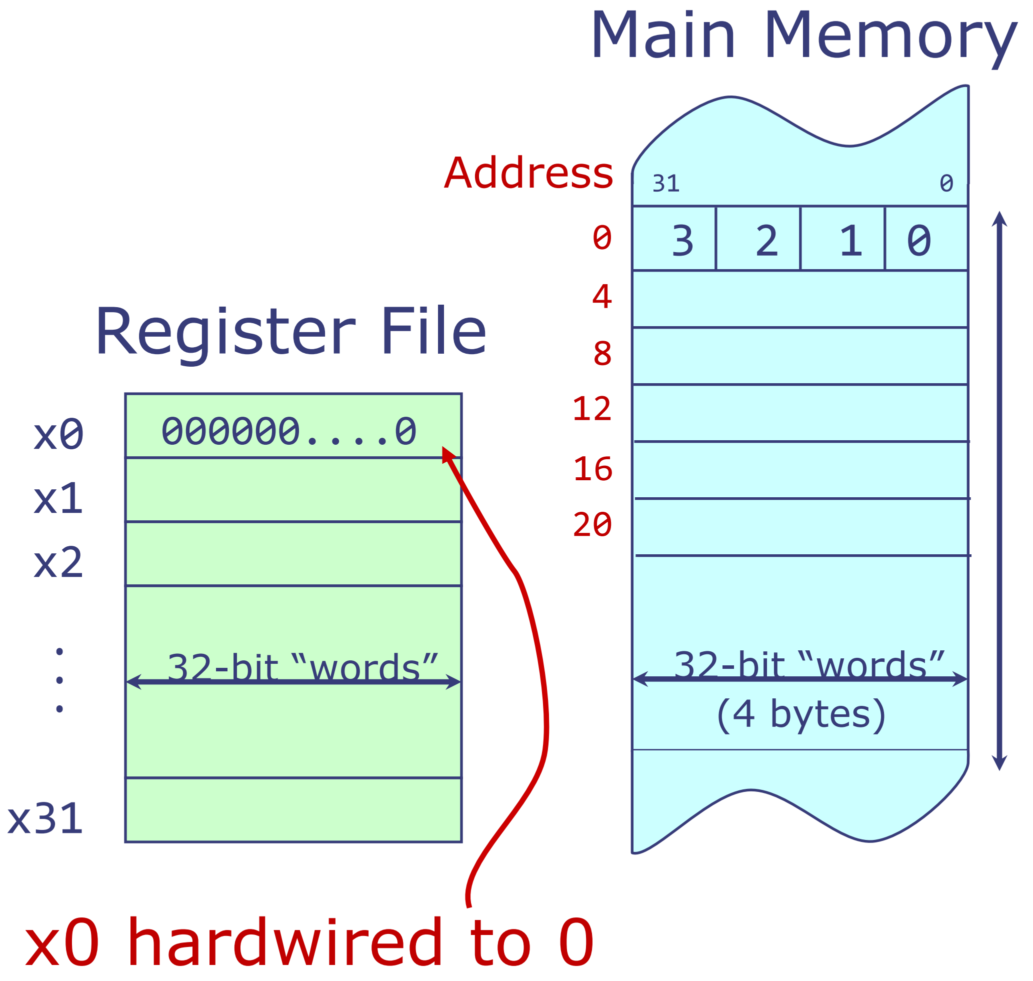
Arithmetic Logic Unit

Machine language directly reflects this structure

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



Registers:

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

Memory:

- Each memory location is 32 bits wide (1 word)
- 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.
  - 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`

$$\begin{array}{r} 5 \\ +\quad 3 \\ \hline \mathit{8} \end{array} \qquad\qquad \begin{array}{r} \mathit{111} \\ 00101 \\ +\quad 00011 \\ \hline \mathit{01000} \end{array}$$

Base 10 — Base 2

  - `sll x3, x1, x2`

    `Shift x1 left`

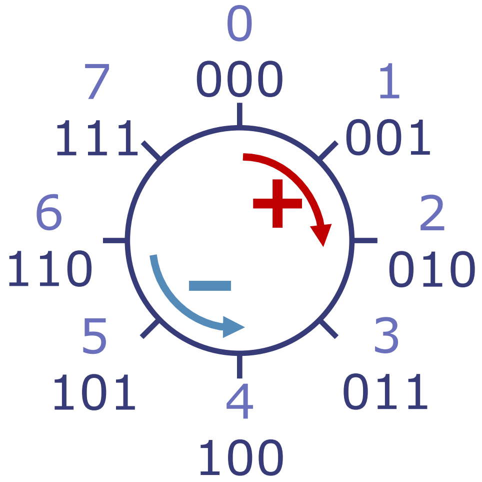
    `by x2 bits`

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

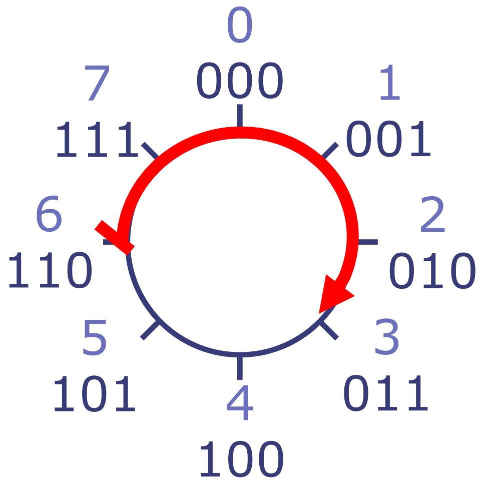
Notice fixed width

Binary Modular Arithmetic

- If we use a **fixed number of bits**, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with mod 2^N
  - Visually, numbers "wrap around":



Example: (6 + 5) mod 2^3 ?



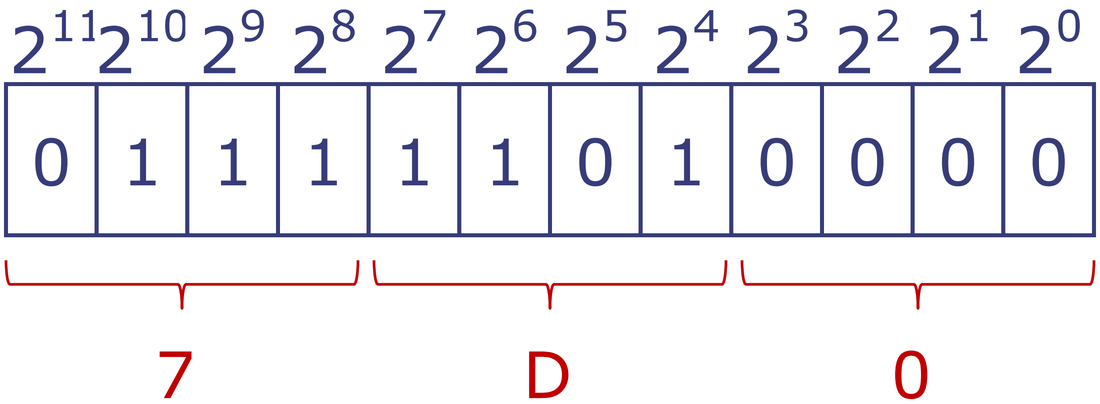
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

Hexadecimal - base 16

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F



2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

7 D 0

0b011111010000 = 0x7D0

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;`

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`
- 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

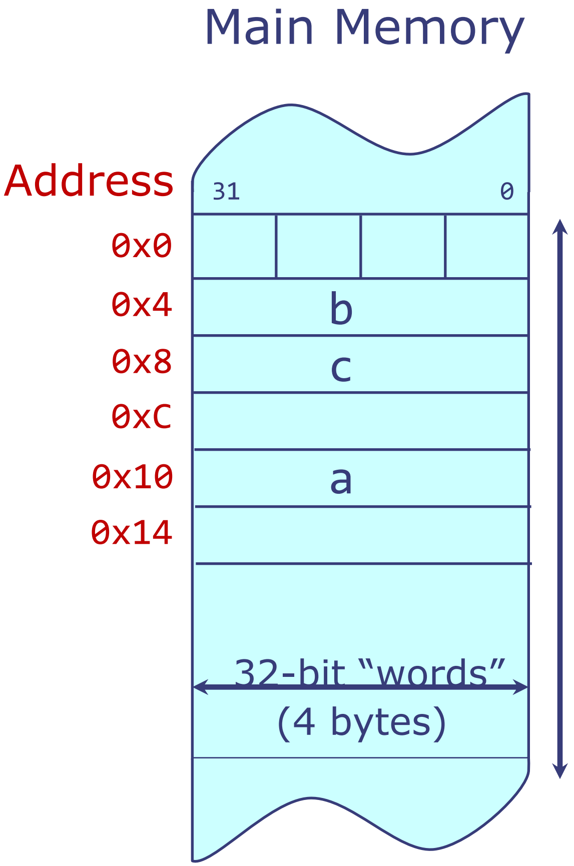
- jal: Unconditional jump and link
  - Example: `jal x3, label`
  - Jump target specified as label
  - label is encoded as an offset from current instruction
  - Link (To be discussed next lecture): is stored in x3
- jalr: Unconditional jump via register and link
  - Example: `jalr x3, 4(x1)`
  - Jump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

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

- For some good technical reasons, RISC-V ISA does not permit us to write memory addresses into instructions directly!
- 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
```

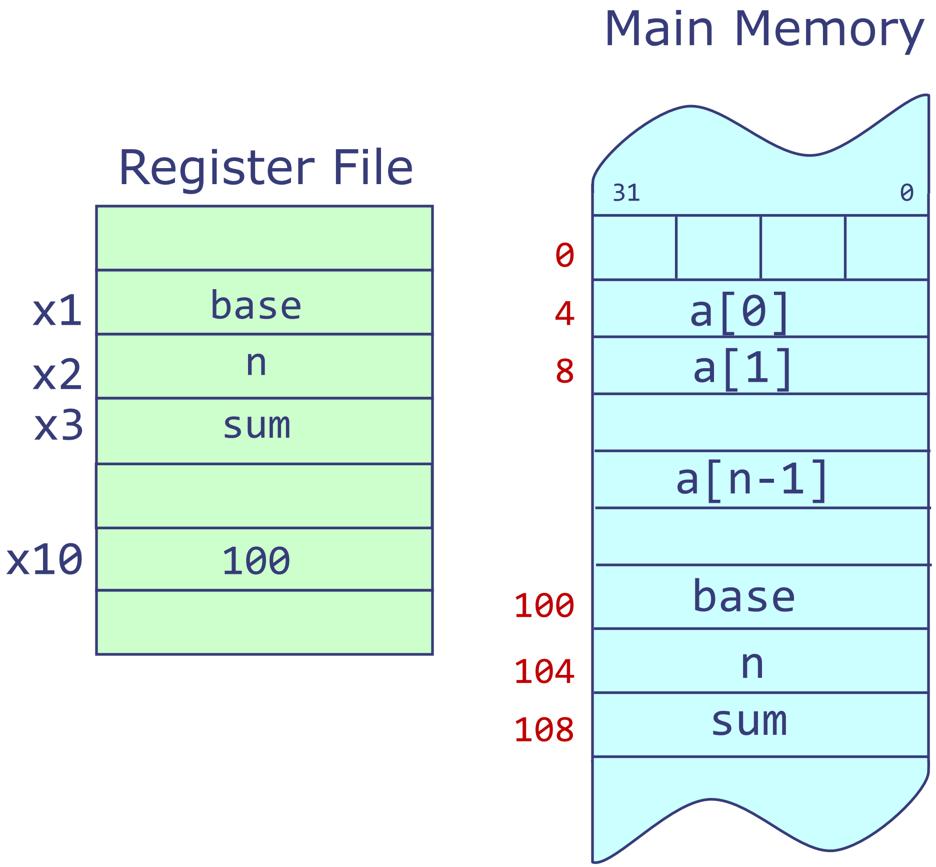
Program to sum array elements

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

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

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

x2 n

x3 sum

x10 100

Main Memory

0

4 a[0]

8 a[1]

a[n-1]

100 base

104 n

108 sum

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 plus a **small** constant.
  - Need to specify 1 destination register (5 bits).
- The constant in the instruction has to be smaller than 12 bits; bigger constants have to be stored in the memory or register and then used explicitly
  - However, small constants are extremely useful in writing programs
- It is because of this restriction we never specify memory addresses directly into instructions

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

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

Encoding Negative Numbers

Sign-Magnitude Representation

We use sign-magnitude representation for decimal numbers, encoding the number’s sign (using “+” and “-”) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

1	1	1	1	1	1	0	1	0	0	0	0

“0” for “+”
“1” for “-”

=-2000

What issues might this encoding have?

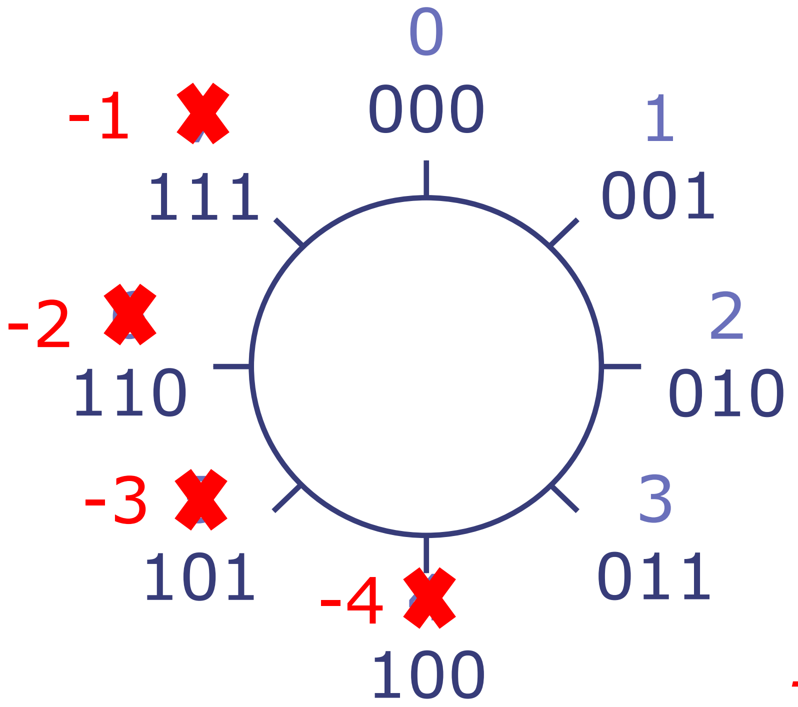
Two representations for 0 (+0, -0)

Circuits for addition and subtraction are different and more complex than with unsigned numbers

Deriving a Better Encoding

Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic?

Yes!

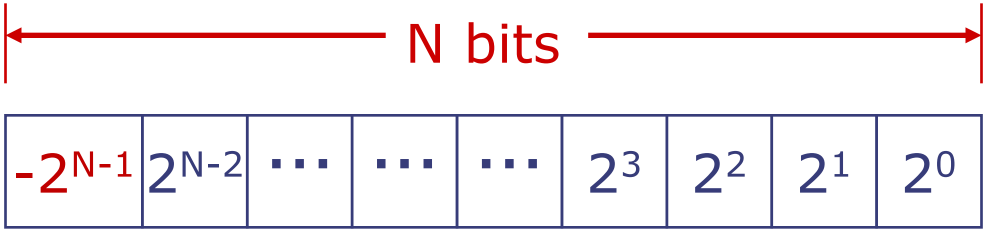


This is called two's complement encoding

Two’s Complement Encoding

In two’s complement encoding, the high-order bit of the N-bit representation has negative weight:

$$v = -2^{N-1} b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i$$



N bits

-2^{N-1}	2^{N-2}	⋯	⋯	⋯	2^3	2^2	2^1	2^0

- Negative numbers have “1” in the high-order bit
- *Most negative number?* 10…0000 *-2^{N-1}*
- *Most positive number?* 01…1111 *$+2^{N-1} - 1$*
- *If all bits are 1?* 11…1111 *-1*

Constants encoded in our instructions use 2’s complement encoding.

Take home

1. Express -4 in 2’s complement notation using 6 bits.
2. Translate the following code snippet into RISC-V assembly:

```
sum = 0
for (i = 0; i <10; i++) begin
    sum = sum + i
end
```

Thank you!

Next lecture:
Implementing Procedures in Assembly