Storing Data in Memory
RV32 So Far...

- Addition/subtraction
  \[
  \text{add } rd, \ rs1, \ rs2 \\
  R[rd] = R[rs1] + R[rs2] \\
  \text{sub } rd, \ rs1, \ rs2 \\
  R[rd] = R[rs1] - R[rs2]
  \]

- Add immediate
  \[
  \text{addi } rd, \ rs1, \ imm \\
  R[rd] = R[rs1] + \text{imm}
  \]
Data Transfer: **Load from** and **Store to** memory

Very fast, but limited space to hold values!

Much larger place to hold values, but slower than registers!
Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a byte (1 word = 4 bytes)
- Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte—least-significant byte (i.e. Little-endian convention)
Memory Addresses are in Bytes

• Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)—works fine if everything is a multiple of 8 bits

• 8 bit chunk is called a *byte* (1 word = 4 bytes)

• Memory addresses are really in *bytes*, not words

• Word addresses are 4 bytes apart
  • Word address is same as address of rightmost byte—least-significant byte (i.e. Little-endian convention)
The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

**Little Endian**

- The order in which BYTES are stored in memory
- Bits always stored as usual (E.g., 0x\text{C2}=0b 1100 0010)

Consider the number 1025 as we typically write it:

- Big Endian
  - ADDR3 ADDR2 ADDR1 ADDR0
  - BYTE0 BYTE1 BYTE2 BYTE3
  - 00000001 00000100 00000000 00000000

**Examples**

- Names in the US (e.g., Bora Nikolić)
- Internet names (e.g., cs.berkeley.edu)
- Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)
- Eating Pizza skinny part first

- Big Endian
  - ADDR3 ADDR2 ADDR1 ADDR0
  - BYTE0 BYTE1 BYTE2 BYTE3
  - 00000001 00000100 00000000 00000000

**Examples**

- Names in China or Hungary (e.g., Nikolić Bora)
- Java Packages: (e.g., org.mypackage.HelloWorld)
- Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)
- Eating Pizza crust first
Data Transfer Instructions
Great Idea #3: Principle of Locality / Memory Hierarchy

- **Processor chip**
  - Extremely fast
  - Extremely expensive
  - Tiny capacity

- **DRAM chip** e.g. DDR3/4/5, HBM/HBM2/3
  - Fast
  - Priced reasonably
  - Medium capacity

- **Random-Access Memory (RAM)**
  - Physical Memory
  - Registers
  - Core
Speed of Registers vs. Memory

• Given that
  • Registers: 32 words (128 Bytes)
  • Memory (DRAM): Billions of bytes
    (2 GB to 64 GB on laptop)
• and physics dictates...
  • Smaller is faster
• How much faster are registers than DRAM??
  • About 50-500 times faster! (in terms of latency of one access - tens of ns)
    • But subsequent words come every few ns
Jim Gray’s Storage Latency Analogy:
How Far Away is the Data?

Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969

100 Memory
1 (ns) Registers
1.5 hr
1 min
My Head
Sacramento
Load from Memory to Register

• C code
  ```c
  int A[100];
  g = h + A[3];
  ```

• Using Load Word (lw) in RISC-V:
  ```asm
  lw x10,12(x15)  # Reg x10 gets A[3]
  add x11,x12,x10  # g = h + A[3]
  ```

Note:  
  x15 – base register (pointer to A[0])
  12 – offset in bytes
  Offset must be a constant known at assembly time
Store from Register to Memory

• C code

```c
int A[100];
```

• Using Store Word (sw) in RISC-V:

```assembly
lw x10,12(x15)  # Temp reg x10 gets A[3]
add x10,x12,x10  # Temp reg x10 gets h + A[3]
```

Note: x15 – base register (pointer)
12,40 – offsets in bytes
x15+12 and x15+40 must be multiples of 4
Loading and Storing Bytes

- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:
  - load byte: lb
  - store byte: sb
- Same format as lw, sw
- E.g., lb x10, 3(x11)
  - contents of memory location with address = sum of “3” + contents of register x11 is copied to the low byte position of register x10.

RISC-V also has “unsigned byte” loads (lbu) which zero extends to fill register. Why no unsigned store byte ‘sbu’?
Example: What is in $x_{12}$?

```
addi x11, x0, 0x3F5
sw x11, 0(x5)
lb x12, 1(x5)
```
Substituting addi

The following two instructions:

```
lw x10,12(x15)  # Temp reg x10 gets A[3]
add x12,x12,x10  # reg x12 = reg x12 + A[3]
```

Replace **addi**:

```
addi x12, value  # value in A[3]
```

But involve a load from memory!

Add immediate is so common that it deserves its own instruction!
Decision Making
• Addition/subtraction
  add rd, rs1, rs2
  sub rd, rs1, rs2

• Add immediate
  addi rd, rs1, imm

• Load/store
  lw  rd,  rs1, imm
  lb  rd,  rs1, imm
  lbu rd,  rs1, imm
  sw rs1, rs2, imm
  sb rs1, rs2, imm
Based on computation, do something different
In programming languages: *if*-statement

RISC-V: *if*-statement instruction is

```
beq reg1, reg2, L1
```

means: go to statement labeled L1 if (value in reg1) == (value in reg2)

….otherwise, go to next statement

*beq* stands for *branch if equal*

Other instruction: *bne* for *branch if not equal*
Types of Branches

• Branch – change of control flow

• Conditional Branch – change control flow depending on outcome of comparison
  • branch if equal (beq) or branch if not equal (bne)
  • Also branch if less than (blt) and branch if greater than or equal (bge)
  • And unsigned versions (bltu, bgeu)

• Unconditional Branch – always branch
  • a RISC-V instruction for this: jump (j), as in j label
Example if Statement

• Assuming translations below, compile if block
  
f → x10   g → x11   h → x12
  i → x13   j → x14

if (i == j)  bne x13,x14,Exit
  f = g + h;  add x10,x11,x12

Exit:

• May need to negate branch condition
Example *if-else* Statement

- Assuming translations below, compile
  
  \[ \begin{align*}
  f & \rightarrow x_{10} \\
  g & \rightarrow x_{11} \\
  h & \rightarrow x_{12} \\
  i & \rightarrow x_{13} \\
  j & \rightarrow x_{14}
  \end{align*} \]

  \[ \text{if (} i = j \text{) } \text{bne } x_{13},x_{14},\text{Else} \]
  
  \[ \text{f} = g + h; \quad \text{add } x_{10},x_{11},x_{12} \]
  
  \[ \text{else} \quad \text{j Exit} \]
  
  \[ \text{f} = g - h; \quad \text{Else:sub } x_{10},x_{11},x_{12} \]
  
  \[ \text{Exit:} \]
Magnitude Compares in RISC-V

• General programs need to test < and > as well.

• RISC-V magnitude-compare branches:
  “Branch on Less Than”
  Syntax: `blt reg1,reg2, Label`
  Meaning: `if (reg1 < reg2) goto Label;`

  “Branch on Less Than Unsigned”
  Syntax: `bltu reg1,reg2, Label`
  Meaning: `if (reg1 < reg2) // treat registers as unsigned integers goto label;`

  Also “Branch on Greater or Equal” `bge` and `bgeu`

Note: No ‘`bgt’ or ‘ble’ instructions
Loops in C/Assembly

- There are three types of loops in C:
  - while
  - do ... while
  - for

- Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.

- Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision-making is conditional branch
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];

C Loop Mapped to RISC-V Assembly

add x9, x1, x0 # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x13, x0, 20 # x13=20
Loop:
bge x11, x13, Done
lw x12, 0(x9) # x12=A[i]
add x10, x10, x12 # sum+=
addi x9, x9, 4 # &A[i+1]
addi x11, x11, 1 # i++
j Loop
Done: