More RISC-V, RISC-V *Functions*

Summary

- RISC Design Principles
  - Smaller is faster: 32 registers, fewer instructions
  - Keep it simple: rigid syntax
- RISC-V Registers: `s0-s11`, `t0-t6`, `x0`
  - No data types, just raw bits, operations determine how they are interpreted
- Memory is byte-addressed
  - no types → no automatic pointer arithmetic

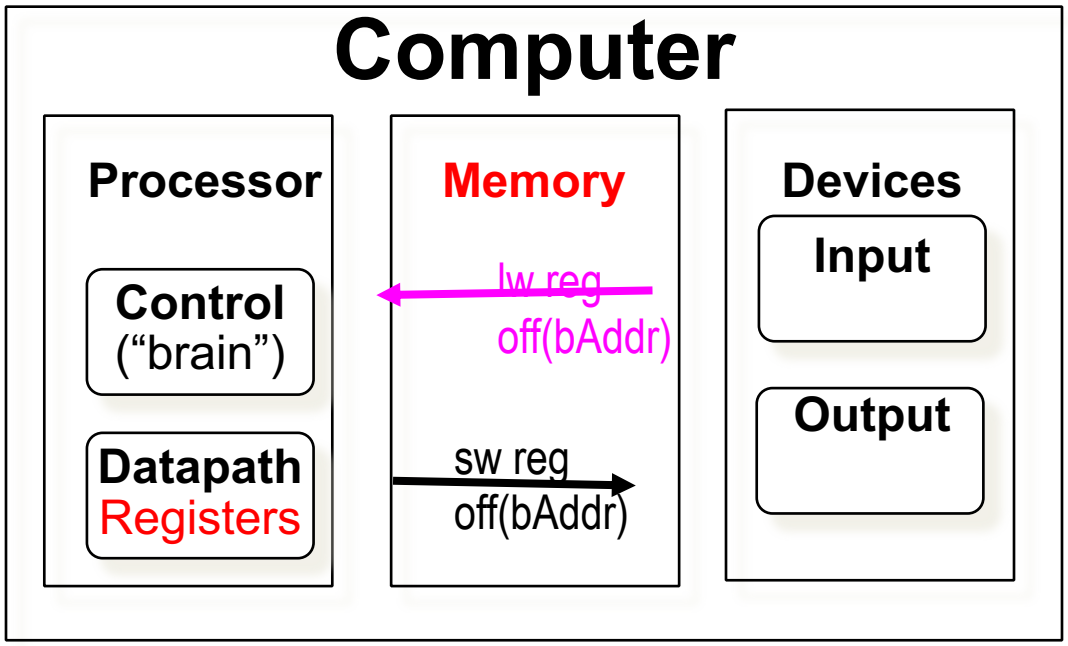
Review of Last Lecture (2/2)

i = "immediate" (constant integer)

- RISC-V Instructions
  - Arithmetic: `add, sub, addi, mult,div`
  - Data Transfer: `lw, sw, lb, sb, lbu`
  - Branching: `beq, bne, j, bge, blt, jal;`
  - Bitwise: `and, or, xor, andi, ori, xori`
  - Shifting: `sll, srl, sra, slli, srli, srai`

Memory and Variable Size

- **So Far:**
  - `lw reg, off(bAddr)`
  - `sw reg, off(bAddr)`



- What about characters (1B) and shorts (sometimes 2B), etc?
- Want to be able to use interact with memory values smaller than a word.

Trading Bytes with Memory (2 approaches)

- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   s0,0(s1)
andi s0,s0,0xFF # lowest byte
```

- Method 2: Load/store byte instructions

```
lb   s1,1(s0)
sb   s1,0(s0)
```

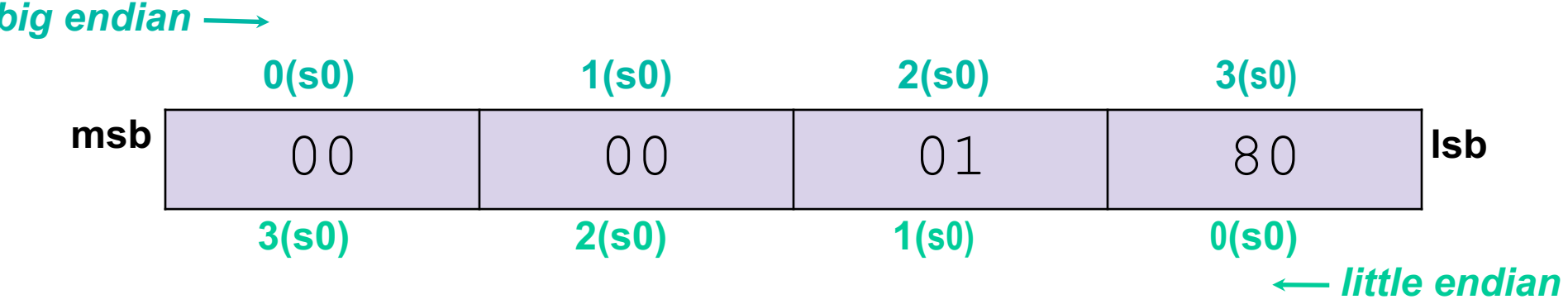
```
*(s0) = 0x00000180
```

00	00	01	80

Endianness

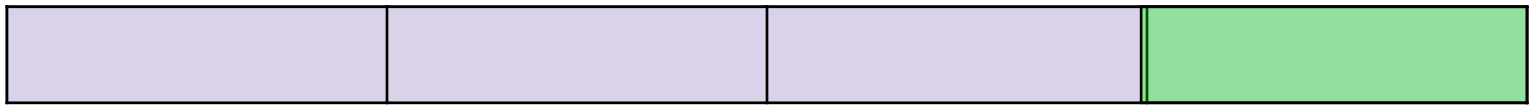
- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte
- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte

```
*(s0) = 0x00000180
```



- RISC-V is Little Endian

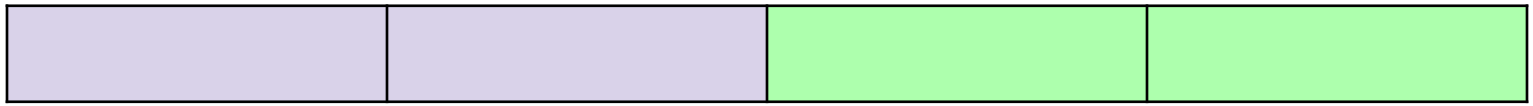
Byte Instructions



- `lb`/`sb` utilize the least significant byte of the register
  - On `sb`, upper 24 bits are ignored
  - On `lb`, upper 24 bits are filled by sign-extension
- For example, let `*(s0) = 0x00000180`:

```
lb s1,1(s0)   # s1=0x00000001
lb s2,0(s0)   # s2=0xFFFFFF80
sb s2,2(s0)   # *(s0)=0x00800180
```

Half-Word Instructions



- `lh reg, off(bAddr)` "load half"
- `sh reg, off(bAddr)` "store half"
  - On `sh`, upper 16 bits are ignored
  - On `lh`, upper 16 bits are filled by sign-extension

Unsigned Instructions

- `lhu reg, off(bAddr)` "load half unsigned"
- `lbu reg, off(bAddr)` "load byte unsigned"
  - On `l(b/h)u`, upper bits are filled by zero-extension
- Why no s(h/b)u? Why no lwu?

Agenda

- More Memory Instructions
- Inequalities
- Pseudo-Instructions
- Bonus: Memory Address Convention
- Bonus: Alternate Register Convention Analogy

Inequalities in RISC-V

- Set Less Than (`slt`)
  - `slt dst,reg1,reg2`
  - if value in `reg1` < value in `reg2`, `dst` = 1, else 0
- Set Less Than Immediate (`slti`)
  - `slti dst,reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0
- These values are all interpreted as signed values
- What if we want to compare unsigned numbers?

Unsigned Inequalities

- Unsigned versions of `slt(i)`:
  - `sltu dst,src1,src2`: unsigned comparison
  - `sltiu dst,src,imm`: unsigned comparison against constant

slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0

- Example:

```
addi  s0,x0,-1     # s0=0xFFFFFFFF
slti  t0,s0,1      # t0=1 (-1 < 1)
sltiu t1,s0,1      # t1=0 (2^32-1 >>> 1)
```

Aside: **RISC-V** Signed vs. Unsigned

- RISC-V terms "signed" and "unsigned" appear in 2 different contexts:
  - Signed vs. unsigned bit extension
    - `lb, lh`
    - `lbu, lhu`
  - Signed vs. unsigned comparison
    - **`slt, slti`**
    - **`sltu, sltiu`**

Question: What C code properly fills in the following blank?

```
do {i--; } while((z = _____________));
```

```
Loop:                   # i→s0, j→s1
addi  s0,s0,-1          # i = i - 1
slti  t0,s1,2           # t0 = (j < 2)
bne   t0,x0 Loop          # goto Loop if t0!=0
slt   t0,s1,s0          # t1 = (j < i)
bne   t0,x0 ,Loop       # goto Loop if t0!=0
```

(A) `j < 2 || j < i`

(B) `j ≥ 2 && j < i`

(C) `j < 2 || j ≥ i`

(D) `j < 2 && j ≥ i`

Question: What C code properly fills in the following blank?

```
do {i--; } while((z = _____________));
```

```
Loop:                   # i→s0, j→s1
addi  s0,s0,-1          # i = i - 1
slti  t0,s1,2           # t0 = (j < 2)
bne   t0,x0 Loop        # goto Loop if t0!=0
slt   t0,s1,s0          # t1 = (j < i)
bne   t0,x0 ,Loop       # goto Loop if t0!=0
```

(A) `j < 2 || j < i`

(B) `j ≥ 2 && j < i`

(C) `j < 2 || j ≥ i`

(D) `j < 2 && j ≥ i`

Agenda

- More Memory Instructions
- Inequalities
- Administrivia
- Pseudo-Instructions
- Bonus: Memory Address Convention
- Bonus: Alternate Register Convention Analogy

Assembler Pseudo-Instructions

- more intuitive for programmers, but NOT directly implemented in hardware
- translated later into instructions which are directly implemented in hardware
- Example:

  `mv dst,reg1` translated into

  `addi dst,reg1,0` or `add dst,reg1,x0`

Full list of RISC-V supported pseudo instructions are on the greensheet

More Pseudo-Instructions

- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
  - translates to: `addi dst x0 imm`
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
  - (translation omitted)

J is a Pseudo-Instruction

- Even the `j` instruction is actually a pseudo-Instruction
  - We will see what this converts to later this lecture
- Pseudo-Instructions are core to writing RISC assembly code and you will see it in any RISC assembly code you read

True Assembly vs RISC-V

- True Assembly Language
  - The instructions a computer understands and executes (directly implemented in hardware!)
- RISC-V Assembly Language
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each RISC-V Assembly instruction becomes 1 or more True Assemblt instructions
- True Assembly $\subset$ RISC-V Assembly