Introduction to Assembly: RISC-V Instruction Set Architecture

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

What is RISC-V?

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
  - Implementors do not pay any royalties
  - But see Amdahl's Law:
    A decent 180 MHz 32b ARM chip costs $6 in quantity
    A Raspberry Pi (with a 1.8 GHz, quad core ARM and everything else) is $35:
    Licensing cost for the ISA can be in the noise
- Appropriate for all levels of computing system, from micro-controllers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants
  - (we’re using 32-bit in class, textbook uses 64-bit)
- Standard maintained by non-profit RISC-V Foundation

Particularly Good For Teaching...

- It is a very very clean RISC
  - No real additional "optimizations"
- Generally only one way to do any particular thing
  - Only exception is two different atomic operation options:
    Load Reserved/Store Conditional
    Atomic swap/add/etc...
- Clean design for efficient concurrent operations
  - Ground-up understanding of how multiple processors can work together
- Kind to implementers
  - Which means relatively kind when we have you implement one!

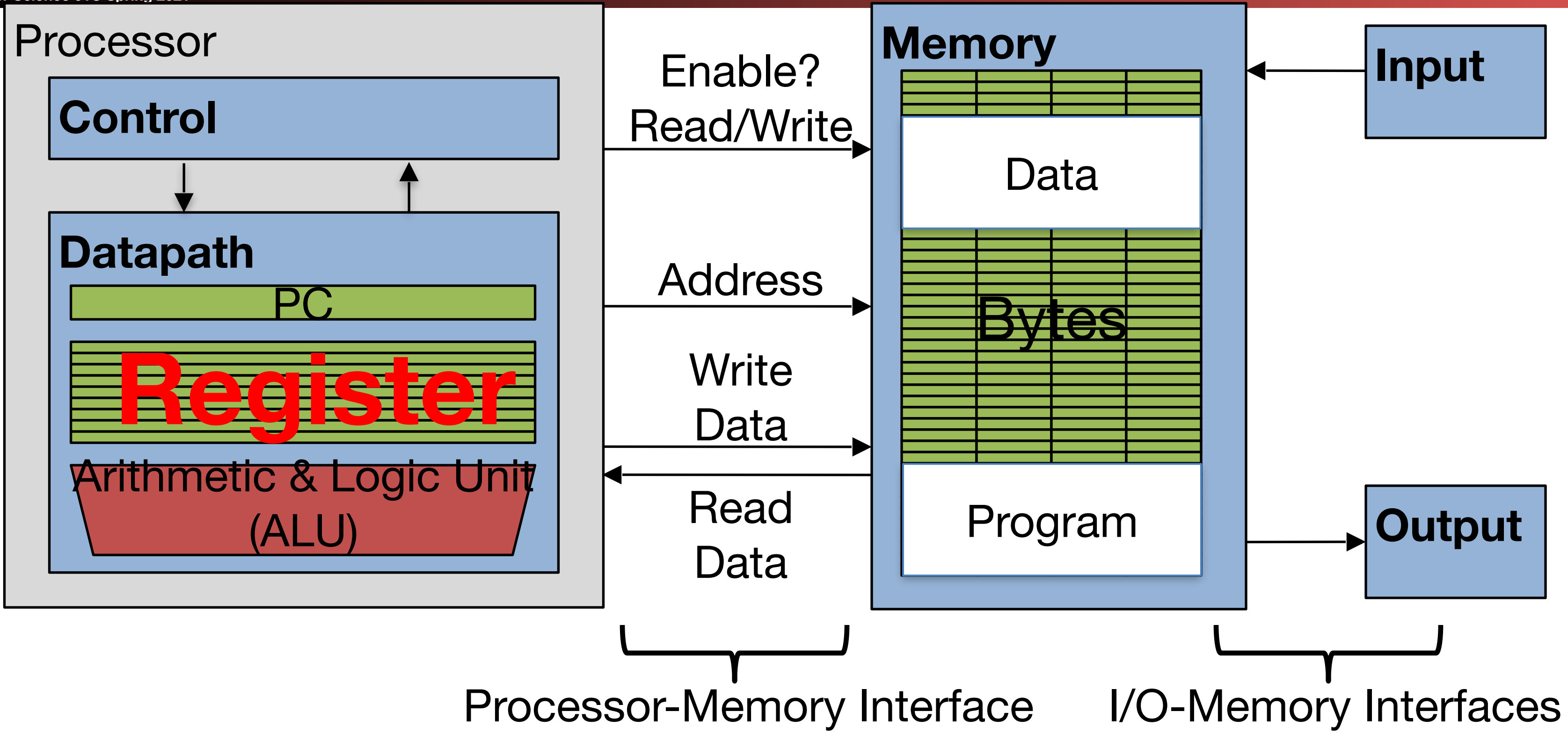
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have variables as you know and love them
    - More primitive, instead what simple CPU hardware can directly support
- Assembly language operands are objects called ***registers***
    - ***Limited number*** of special places to hold values, built directly into the hardware
    - Arithmetic operations can only be performed on these in a RISC!
        - Only memory actions are loads & stores
        - CISC can also perform operations on things ***pointed to*** by registers
- Benefit:
    - Since registers are directly in hardware, they are very fast to access

Registers live inside the Processor



Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 16 GB on laptop)
- and physics dictates…
  - Smaller is faster
- How much faster are registers than DRAM??
- About 100-500 times faster!
  - in terms of ***latency*** of one access

Number of RISC-V Registers

- Drawback: Registers are in hardware. To keep them really fast, their number is limited:
  - Solution: RISC-V code must be carefully written to use registers efficiently
- 32 registers in RISC-V, referred to by number x0 – x31
  - Registers are also given symbolic names:
    These will be described later and are a "convention"/"ABI" (Application Binary Interface):
    Not actually enforced in hardware but needed to follow to keep things consistent
  - Why 32? Smaller is faster, but too small is bad.
    - Plus need to be able to specify 3 registers in operations...
  - Each RISC-V register is 32 bits wide (RV32 variant of RISC-V ISA)
  - Groups of 32 bits called a word in RISC-V ISA
  - P&H CoD textbook uses the 64-bit variant RV64 (explain differences later)
- x0 is special, always holds the value zero
  - So really only 31 registers able to hold variable values

C, Java Variables vs. Registers

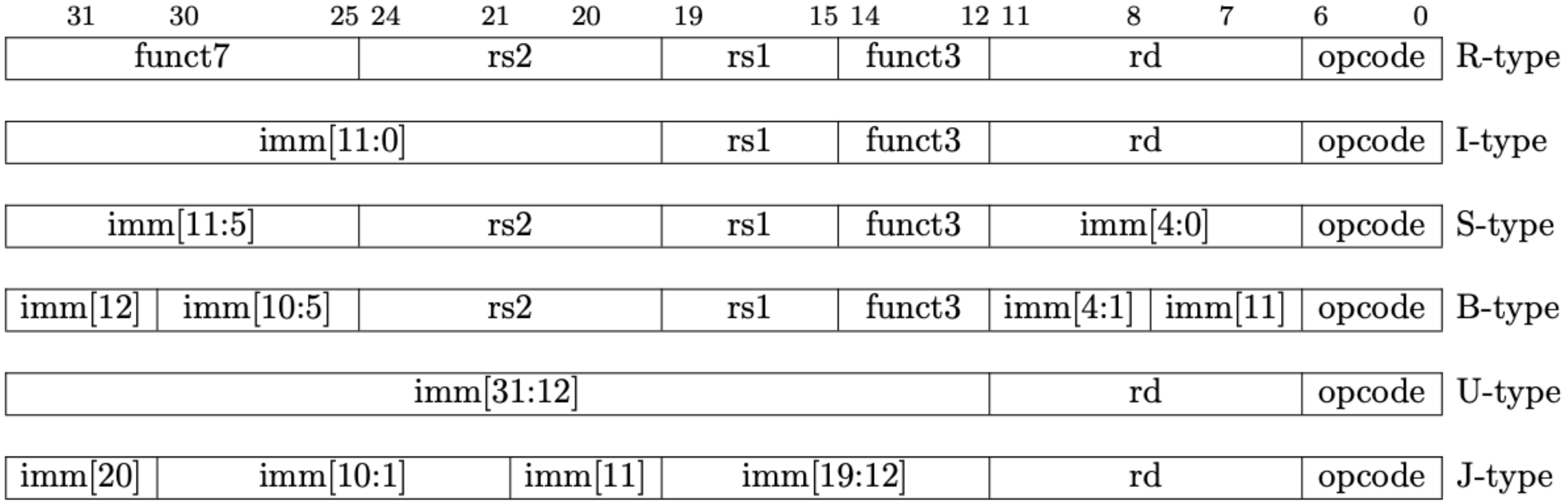
- In C (and most HLLs):
  - Variables declared and given a type
    - Example:
      ```
      int fahr, celsius;
      char a, b, c, d, e;
      ```
  - Each variable can ONLY represent a value of the type it was declared (e.g., cannot mix and match int and char variables)
    - If types are not declared, the object carries around the type with it. EG in python:
      ```
      a = "fubar" # now a is a string
      a = 121 # now a is an integer
      ```
- In Assembly Language:
  - Registers have ***no type***;
  - Operation determines how register contents are interpreted

RISC-V Memory Alignment...

- RISC-V does not ***require*** that integers be word aligned...
  - But it is very ***very bad*** if you don't make sure they are...
- Consequences of unaligned integers
  - Slowdown: The processor is allowed to be a lot slower when it happens
    - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in ***software***!
      An unaligned load could take ***hundreds of times longer***!
  - Lack of ***atomicity***: The whole thing doesn't happen at once...
    can introduce lots of very subtle bugs
- So in ***practice***, RISC-V requires integers to be aligned on 4-byte boundaries

RISC-V Instructions

- Instructions are fixed, 32b long
  - Must also be word aligned, or half-word aligned if the 16b optional (C) instruction set is also enabled
- Only a few formats (we'll go into detail later)...



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

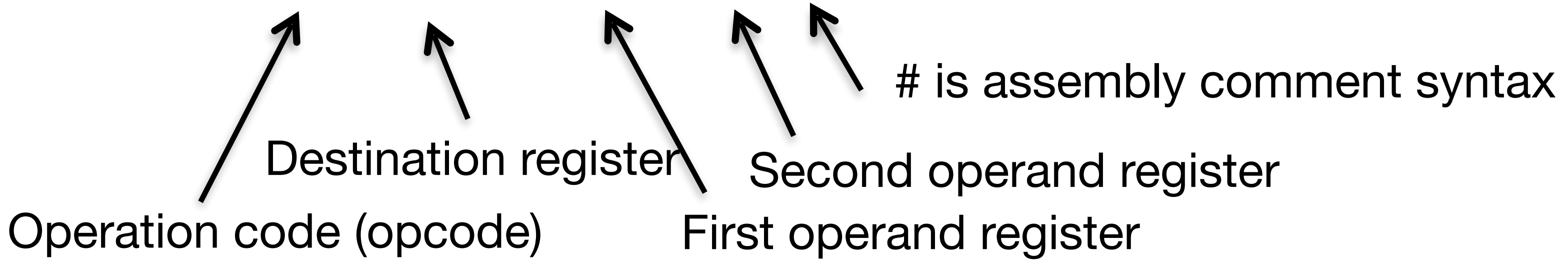
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

RISC-V Instruction Assembly Syntax

- Instructions have an opcode and operands

E.g., **`add x1, x2, x3 # x1 = x2 + x3`**



Addition and Subtraction of Integers

- Addition in Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: `a = b + c` (in C)

    where C variables ⇔ RISC-V registers are:

    `a ⇔ x1, b ⇔ x2, c ⇔ x3`
- Subtraction in Assembly
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: `d = e - f` (in C)

    where C variables ⇔ RISC-V registers are:

    `d ⇔ x3, e ⇔ x4, f ⇔ x5`

No-Op

- A No-op is an instruction that does nothing...
    - Why?
      You may need to replace code later: No-ops can fill space, align data, and perform other options
- By ***convention*** RISC-V has a specific no-op instruction...
    - **`add x0 x0 x0`**
- Why?
    - Writes to x0 are always ignored...
      RISC-V uses that a lot as we will see in the jump-and-link operations
    - Making a "standard" no-op improves the disassembler and can potentially improve the processor
        - Special case the particular conventional no-op.

Addition and Subtraction of Integers Example 1

- How to do the following C statement?

```
a = b + c + d - e;
```

- Break into multiple instructions

```
add x1, x2, x3   # temp = b + c
add x1, x1, x4   # temp = temp + d
sub x1, x1, x5   # a = temp - e
```

- A single line of C may turn into several RISC-V instructions

`add x3,x4,x0` (in RISC-V) same
`f = g` (in C)

Immediates

- *Immediates are used to provide numerical constants*
- Constants appear often in code, so there are special instructions for them:
- Ex: Add Immediate:

  `addi x3,x4,-10` (in RISC-V)

  `f = g - 10` (in C)

  where RISC-V registers `x3,x4` are associated with C variables **f, g**
- Syntax similar to `add` instruction, except that last argument is a number instead of a register

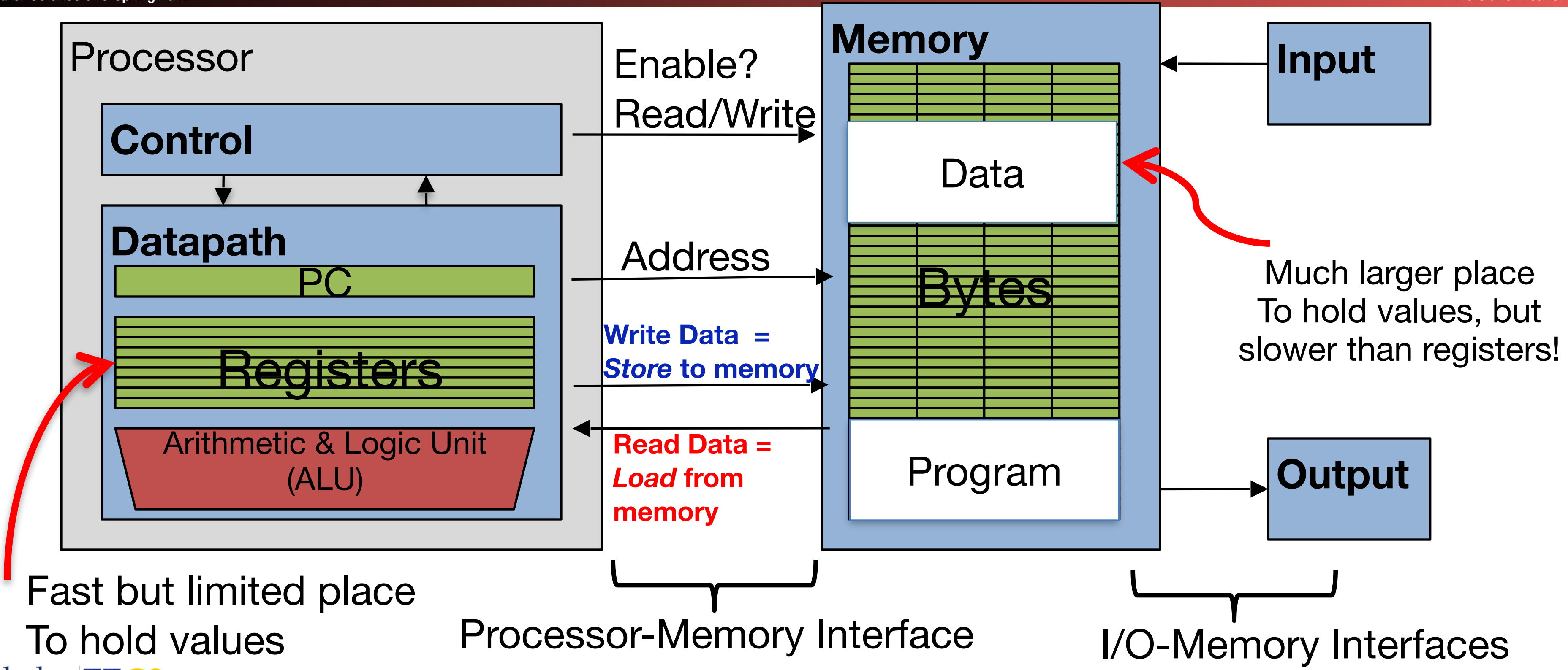
  `addi x3,x4,0` (in RISC-V) same as

  `f = g` (in C)

Immediates & Sign Extension...

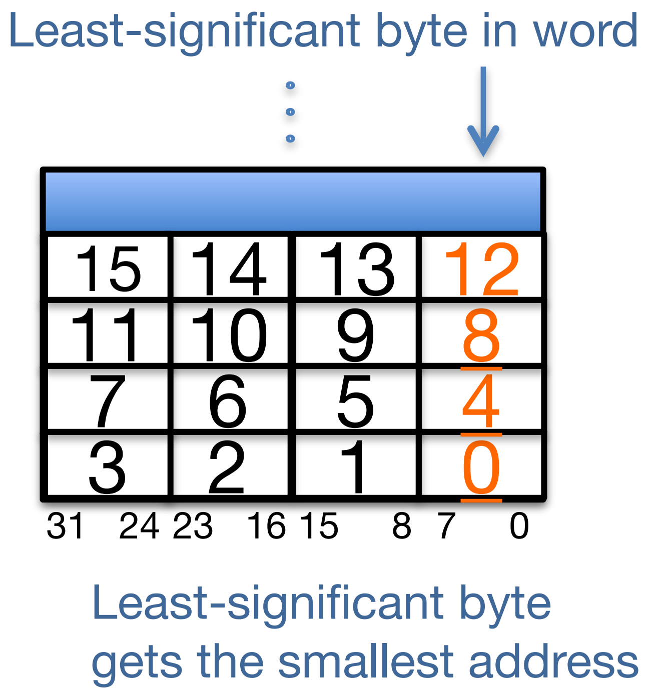
- Immediates are necessarily small
  - An I-type instruction can only have 12 bits of immediate
- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the largest bit
- So for a 12b immediate...
  - Bits 31:12 get the same value as Bit 11

Data Transfer:
Load from and Store to memory


Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data = *Store* to memory
Read Data = *Load* from memory
Memory
Data
Bytes
Program
Input
Output
Much larger place To hold values, but slower than registers!
Fast but limited place To hold values
Processor-Memory Interface
I/O-Memory Interfaces

Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)
  - So everything is a multiple of 8 bits
- Remember, 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)



Least-significant byte in word
15 14 13 12
11 10 9 8
7 6 5 4
3 2 1 0
31 24 23 16 15 8 7 0
Least-significant byte gets the smallest address

Transfer from Memory to Register

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

- Using Load Word (`lw`) in RISC-V:
  ```
  lw   x10,12(x13)  # Reg x10 gets A[3]
  add  x11,x12,x10  # g = h + A[3]
  ```

Assume: `x13` – base register (pointer to A[0])

Note: `12` – offset in bytes

Offset must be a constant known at ***assembly time***

Transfer from Register to Memory

- C code

```
int  A[100];
A[10] = h + A[3];
```

- Using Store Word (`sw`) in RISC-V:

```
lw   x10,12(x13)   # Temp reg x10 gets A[3]
add  x10,x12,x10   # Temp reg x10 gets h + A[3]
sw   x10,40(x13)   # A[10] = h + A[3]
```

Assume: `x13` – base register (pointer)

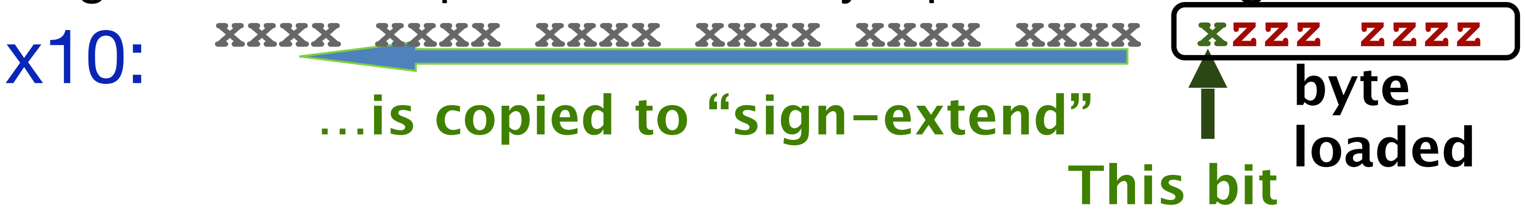
Note: `12,40` – offsets in bytes

x13+12 and x13+40 must be multiples of 4 to maintain alignment

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 extend to fill register. Why no unsigned store byte **`sbu`**?


x10:
xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz
…is copied to "sign-extend"
This bit
byte loaded

Example - Tracing Assembly Code

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

What's the value in x12?

Answer	x12
A	0x5
B	0xf
C	0x3
D	0xffffffff

Example - Tracing Assembly Code

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

What's the value in x12?

Answer	x12
A	0x5
B	0xf
[✓ C]	[✓ 0x3]
D	0xffffffff

Note Endianness...

- Remember, RISC-V is "little endian"
  - byte[0] = least significant byte of the number
  - byte[3] = most significant byte of the number
- So for this example...
  - `byte[0] = 0xf5`
  - `byte[1] = 0x03`
  - `byte[2] = 0x00`
  - `byte[3] = 0x00`

Another Example

```
addi x11,x0,0x8f5
sw x11,0(x5)
lb x12,1(x5)
```

What's the value in x12?

Answer	x12
A	0x8
B	0xf8
C	0x3
D	0xfffffff8

Example - Tracing Assembly Code

```
addi x11,x0,0x8f5
sw x11,0(x5)
lb x12,1(x5)
```

What's the value in x12?

Answer	x12
A	0x8
B	0xf8
C	0x3
D	0xfffffff8

Two Reasons for The Answer...

- The immediate got sign extended...
  - So **`0xffff8f5`** got written
- Then load byte is called
  - So it will load byte[1], which is **`0xf8`**
- But load byte sign extends too...
  - So what gets loaded into the register is **`0xfffffff8`**
- If we did **`lbu`** we'd instead get **`0xf8`**

RISC-V Logical Instructions

Useful to operate on fields of bits within a word

e.g., characters within a word (8 bits)

Operations to pack /unpack bits into words

Called logical operations

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit XOR	^	^	**`xor`**
Shift left logical	<<	<<	**`sll`**
Shift right	>>	>>	**`srl/sra`**

Logical Shifting

- Shift Left Logical: `slli x11,x12,2` # x11 = x12<<2
  - Store in x11 the value from x12 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C

  Before: $0000\ 0002_{hex}$

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

  After: $0000\ 000\underline{8}_{hex}$

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 10\underline{00}_{two}$

  What arithmetic effect does shift left have?

- Shift Right Logical: `srli` is opposite shift; >>
  - Zero bits inserted at left of word, right bits shifted off end

Arithmetic Shifting

- *Shift right arithmetic* (**`srai`**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$
- If execute sra x10, x10, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Transfer Array Value from Memory to Register With Variable Indexing

- C code

```
int A[100];/* x13 */
int i;       /* x14 */
...
g = h + A[3]; /* h = x12, g = x11, tmp = x15 */
```

- Using Load Word (`lw`) in RISC-V with pointer arithmatic:

```
sll x15,x14,2   /* Multiply by 4 for ints */
add x15,x15,x13 /* A + 4 * i */
lw  x10,0(x15)
add x11,x12,x10
```

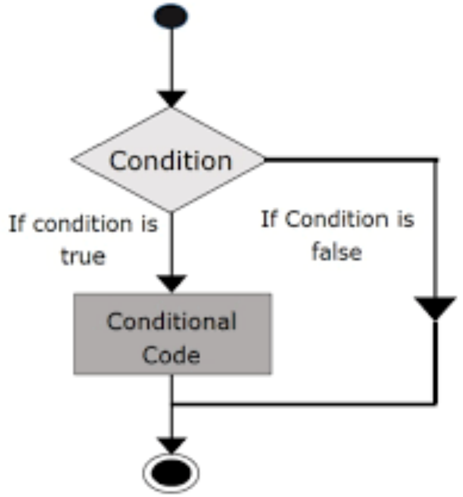
Computer Decision Making

- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for programming languages: *if*-statement

- RISC-V: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to instruction labeled L1
  if (value in register1) == (value in register2)
  ….otherwise, go to next instruction
- **`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`**)

- **Unconditional Branch** – always branch
  - a RISC-V instruction for this*: jump (`j`)*
    - *We will see later that j doesn't exist (it's a "pseudo-instruction")*

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

Labels In Assembly Language...

- We commonly see "labels" in the code
  - `foo: add x2 x1 x0`
- The assembler converts these into positions in the code
  - What address in the code that label is at...
- Then when you have control flow instructions like jumps and branches...
  - e.g. `bne x0 x2 foo`
- The assembler in outputting the code does the necessary math so the jump or branch will go to the right place

Example *if* Statement

- Assuming assignments below, compile *if* block

  f → `x10` g → `x11` h → `x12`

  i → `x13` j → `x14`

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

Example *if-else* Statement

- Assuming assignments below, compile

  f → `x10`   g → `x11`   h → `x12`   i → `x13`   j → `x14`

```
if (i == j)                    bne x13,x14,else
   f = g + h;                  add x10,x11,x12
else                           j done
   f = g - h;          else:   sub x10,x11,x12
                       done:
```

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C); 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) // Registers are signed
                goto label;

- "Branch on Less Than Unsigned"

  Syntax: **`bltu reg1,reg2, label`**

  Meaning: if (reg1 < reg2) // treat registers as unsigned integers
      goto label;

"Branch on Greater Than or Equal" (and it's unsigned version) also exists.

But RISC philosophy...

- A CISC might also have "branch if greater than"...
  - But RISC-V doesn't.
- Instead you can switch the argument
  - branch if greater then reg1 reg2...
  - branch if less than reg2 reg1
- So we have pseudo-instructions, where the assembler converts things
  - **`bgt x2 x3 foo`** becomes
  - **`blt x3 x2 foo`**

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
   sum += A[i];
```

```
# Assume x8 holds pointer to A
# Assign x10=sum, x11=i
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x12,x0,20  # x12=20
Loop:
bge x11, x12, exit:
sll x13, x11, 2   # i * 4
add x13, x13, x8  # & of A + i
lw x13, 0(x13)    # *(A + i)
add x10, x10, x13 # increment sum
addi x11, x11, 1  # i++
j Loop            # Iterate
exit:
```

Comments...

- The simple translation is suboptimal!
    - A more efficient way:
- Inner loop is now 4 instructions rather than 7
    - And only 1 branch/jump rather than two:
      Because first time through is always true so can move check to the end!
- The compiler will often do this automatically for optimization
    - See that i is only used as an index in a loop

```
# Assume x8 holds pointer to A
# Assign x10=sum
add  x10, x0, x0  # sum=0
add  x11, x0, x8  # Copy of A
addi x12,x11, 80  # x12=80 + A
loop:
lw   x13, 0(x11)
add  x10, x10, x13
addi x11, x11, 4
blt  x11, x12, loop
```

And Premature Optimization...

- In general we want ***correct*** translations of C to RISC-V
- It is ***not*** necessary to optimize
    - Just translate each C statement on its own
- Why?
    - Correctness first, performance second
        - Getting the wrong answer fast is not what we want from you...
    - We're going to need to read your assembly to grade it!
        - Multiple ways to optimize, but the straightforward translation is mostly unique-ish.

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

In Conclusion,…

- Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
- Hardware registers provide a few very fast variables for instructions to operate on
- RISC-V ISA requires software to break complex operations into a string of simple instructions, but enables faster, simple hardware
- Assembly code is human-readable version of computer's native machine code, converted to binary by an *assembler*