

# **Introduction to Assembly: RISC-V Instruction Set Architecture**

# Administrivia

- Assignments Due Next Week:
  - Homework 2: 9/22
  - Lab 2: 9/17 (today!)
  - Lab checkoffs will end *promptly* at 4PM on Fridays!
- Project 1 is due on 9/20
- Upcoming Assignments:
  - Lab 3, due 9/24
  - Homework 3 released, due 9/24

# Outline

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

# Outline

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

# Levels of Representation/Interpretation

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

High Level Language  
Program (e.g., C)

*Compiler*

Assembly Language  
Program (e.g., RISC-V)

*Assembler*

Machine Language  
Program (RISC-V)

*Machine  
Interpretation*

Hardware Architecture Description  
(e.g., block diagrams)

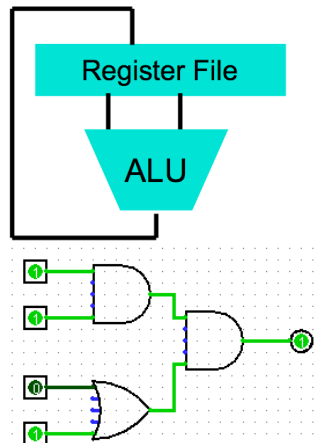
*Architecture  
Implementation*

Logic Circuit Description  
(Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Anything can be represented  
as a *number*,  
i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



# Instruction Set Architecture (ISA)

- Job of a CPU (Central Processing Unit, aka Core): execute instructions
- Instructions: CPU's primitive operations
  - Instructions performed one after another in sequence
  - Each instruction does a small amount of work (a tiny part of a larger program).
  - Each instruction has an operation applied to operands,
  - and might be used to change the sequence of instructions.
- CPUs belong to “families,” each implementing its own set of instructions
- CPU's particular set of instructions implements an Instruction Set Architecture (ISA)
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), x86\_64, ...

# Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations,  
Complex Instruction Set Computer (CISC)

- VAX architecture had an instruction to evaluate polynomials!

- RISC philosophy: Cocke IBM, Patterson (UCB), Hennessy (Stanford), 1980s  
Reduced Instruction Set Computer (RISC)

- Keep the instruction set small and simple, makes it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones

# So Why Do Some Architectures “Win”?

- The big winners: x86/x64 (servers) and Arm (phones/embedded)
  - Neither are the cheapest nor the best architectures available...
- They won because of the legacy software stack...
  - x86 had Windows and then Linux for servers and a history of optimizing for performance ***without breaking old things***.
    - For a decades everything automatically ran faster because of Moore’s Law ...
  - Arm became entrenched with Linux->Android in the phone market
- But since ***our*** focus is understanding how computers work, our software stack is RISC-V



# Assembly Language Programming

ARM

```
LDR r0,[p_a]
LDR r1,[p_b]
ADD r3,r0,r1
STR r3,[p_w]
LDR r2,[p_c]
ADD r0,r2,r3
STR r0,[p_x]
LDR r0,[p_d]
ADD r3,r2,r0
STR r3,[p_y]
```

k and Weaver

- Each assembly language is tied to a particular ISA (its just a human readable version of machine language).
- Why program in assembly language versus a high-level language?
  - Back in the day, when ISAs were complex and compilers were immature .... hand optimized assembly code could beat what the compiler could generate.
- These days ISAs are simple and compilers beat humans
  - Assembly language still used in small parts of the OS kernel to access special hardware resources
- For us ... learn to program in assembly language
  - Best way to understand what compilers do to generate machine code
  - Best way to understand what the CPU hardware does

x86

```
pushl %ebp
movl %esp,%ebp
subl $0x4,%esp
movl $0x0,0xffffffff(%ebp)
cmpl $0x63,0xffffffff(%ebp)
jle 08048930
jmp 08048948
```



# And the Road To Future Classes...

- CS164: Compilers
  - Learn how to build compilers. A compiler goes from source code to assembly language.
- CS162: O/S
  - OS needs a small amount of assembly for doing things the "high level" language doesn't support
    - Such as accessing special resources
- CS152: Computer Architecture
  - How to build the hardware that supports the assembly:  
So we use assembly to debug the hardware design!
- CS161: Security
  - Exploit code ("shell code") is often in assembly and exploitation often requires understanding the assembly language & calling-convention of the target

# 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
  - Large community of users [riscv.org](https://riscv.org): industry, academia
  - Full software stack
- 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 well designed RISC (the 5<sup>th</sup> generation) - informed from earlier attempts
- 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!

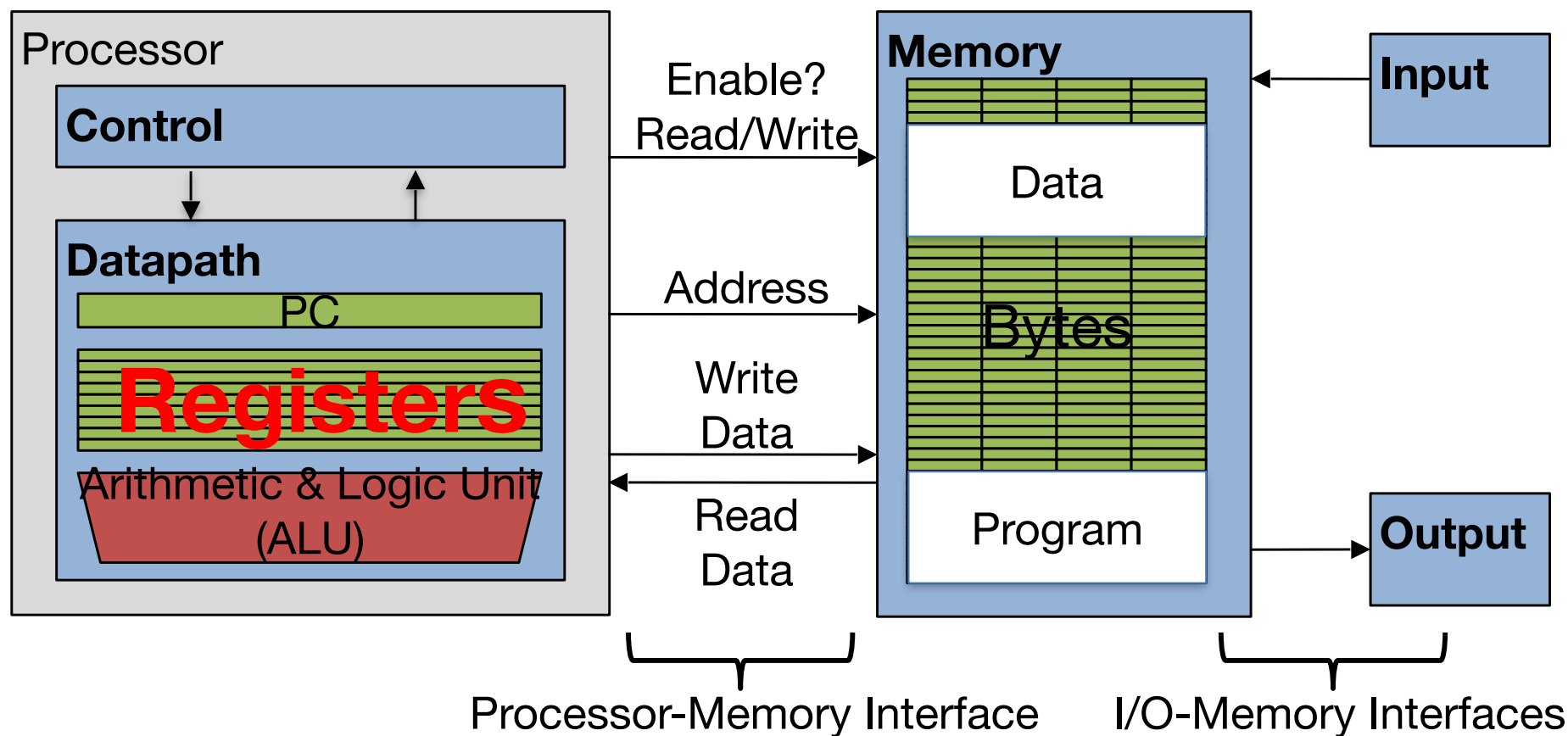
# Outline

- 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: *instructions to move values from memory to registers, instructions to operation on registers*





# 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

“Regfile”



- Drawback: The number of registers is limited (32 on RISC-V)
- Why limited number?
  - Registers are in hardware. To keep them really fast, their number is limited.
  - Limited number of bits in instructions to be allocated to indexing/addressing registers.
- 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 software consistent
  - 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 and can't be changed
  - So really only 31 registers able to hold variable values

# C, Java Variables vs. Registers

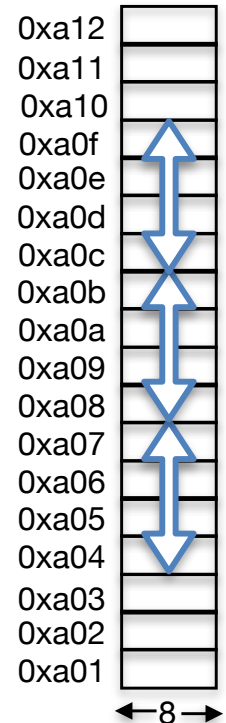
- In C (and most HLLs):
  - Variables declared and given a type
    - Example:

```
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)
    - In some languages (eg., Python) If types are not declared, the object carries around the type with it:

```
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

# A word about RISC-V Memory Alignment...

- Memory is addressed by Bytes, but many RISC-V memory operations address 32-bit words
- Word-aligned: integers start on even **4-Byte boundaries** (address is even multiple of 4 - last 2-bits of address is 00)
- 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 word-aligned



# RISC-V Instructions

- Instructions are fixed, 32b long
  - *Must be word aligned*
- Instruction *formats* define how machine instructions are encoded.
- Each instruction uses one of these predefined formats:

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			

- More later ...

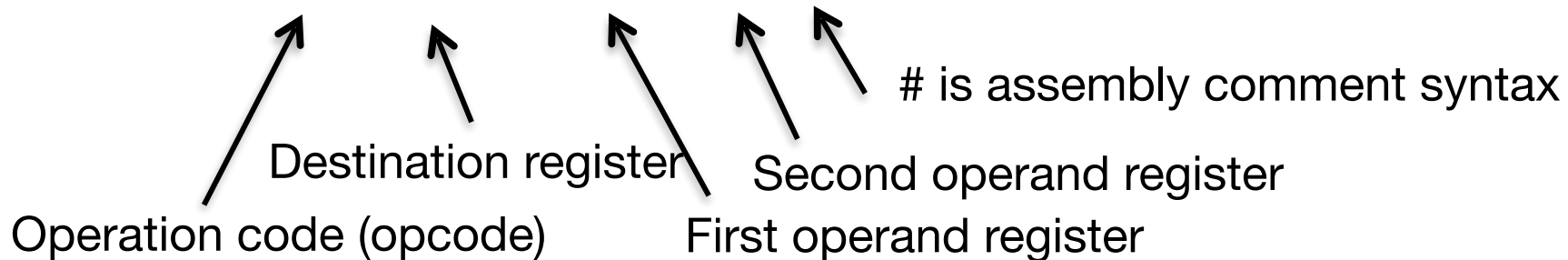
# Outline

- 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  $\Leftrightarrow$  RISC-V registers are:

$$a \Leftrightarrow x1, b \Leftrightarrow x2, c \Leftrightarrow x3$$

- Subtraction in Assembly

- Example: **sub x3, x4, x5** (in RISC-V)
- Equivalent to: **d = e - f** (in C)  
where C variables  $\Leftrightarrow$  RISC-V registers are:

$$d \Leftrightarrow x3, e \Leftrightarrow x4, f \Leftrightarrow x5$$



# 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
```

# Register x0

**Very useful:** *always holds zero and can never be changed  
(does not require initialization)*

- Ex: Moving a value from one register to another:

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

- Or, whenever a value is produced and we want to throw it away (in the “bit bucket), write to x0:
- By **convention** RISC-V has a specific no-op instruction  
`add x0 x0 x0`
- Also, we will see x0 used later with “jump-and-link” instruction

# 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)

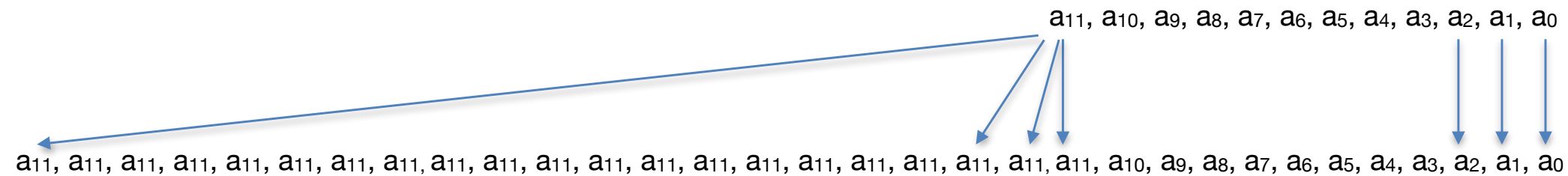
where RISC-V registers `x3,x4` are associated with C variables `f, g`

- Syntax similar to `add` instruction, except that last argument is a number instead of a register

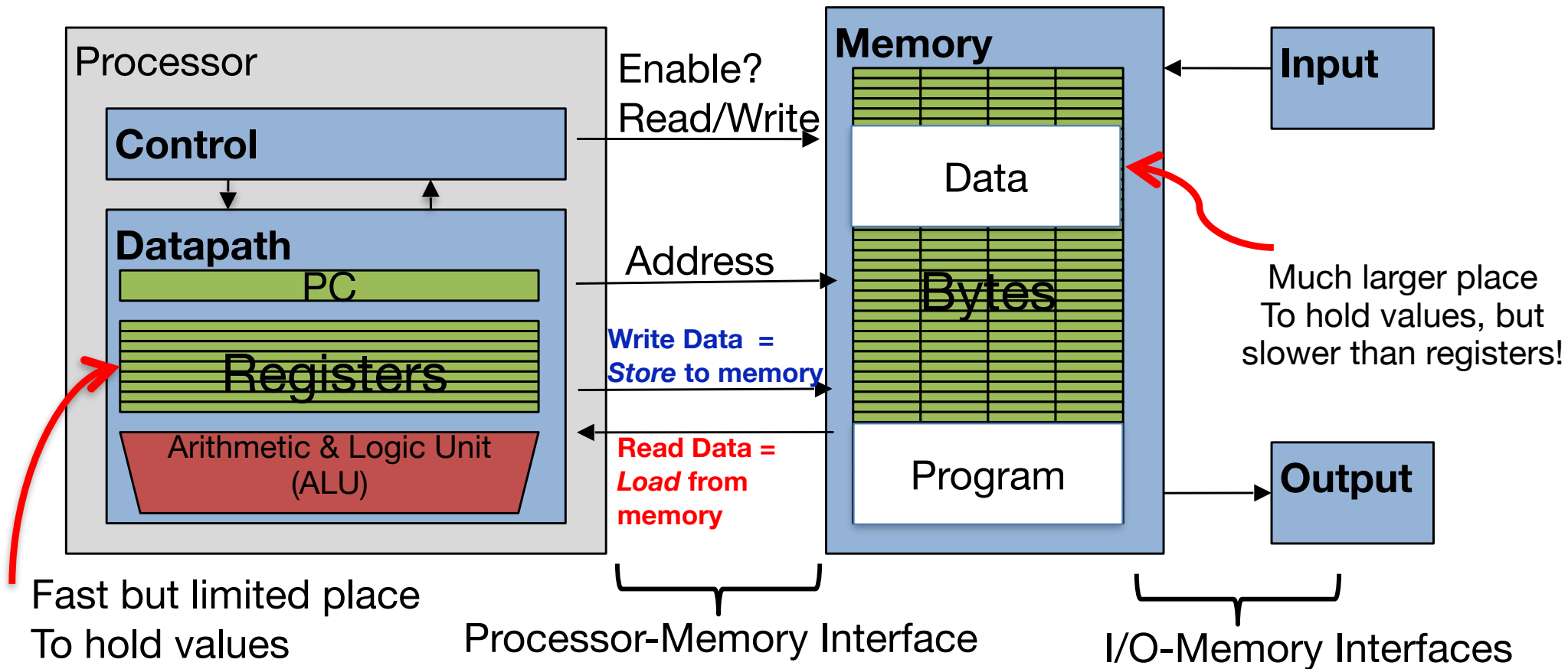
`addi x3,x4,0` (in RISC-V) same as  
`f = g` (in C)

# 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 top bit
- So for a 12b immediate...
  - Bits 31:12 get the same value as Bit 11



# Data Transfer: Load from and Store to memory



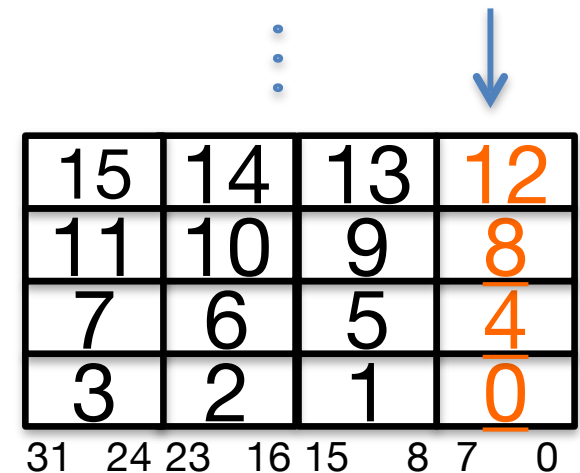
# Memory Addresses are in Bytes

- 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, size of word is 4 bytes
- Memory is addressable to individual bytes

∴ Word addresses are 4 bytes apart

- words take on the address of their least-significant byte (in “little-endian convention”)
- remember to keep words aligned

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];
```

Assume: **x13** holds base register (pointer to A[0]), **x12** holds **h**

Note: **12** is offset in bytes

Offset must be a constant known at *assembly time*

- Using Load Word (**lw**) in RISC-V:

```
lw  x10,12(x13) # reg x10 gets A[3]  
add x11,x12,x10 # g = h + A[3]
```

# Transfer from Register to Memory

- C code

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

Assume: **x13** holds base register (pointer), **x12** holds **h**

Note: **12, 40** is offsets in bytes

- 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]
```

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



# Loading and Storing Bytes

- nsfers:
- RISC-V also has “unsigned byte” loads (**lbu**) which zero extend to fill register. Why no unsigned store byte **sbu**?

x10: xxxxx xxxxx xxxxx xxxxx xxxxx xxxxx

...is copied to "sign-extend"

zxxx zxxx

byte loaded

This bit

# Example - Tracing Assembly Code

```
addi x11, x0, 0x3f5  
sw x11, 0(x5) 00 00 03 f5  
lb x12, 1(x5)
```

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

What's the final value in x12?

# Example - Tracing Assembly Code

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

*Handwritten notes:*  
0011 1110 101  
No sign extension

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)**

*Handwritten notes:*  
MSB L8B  
x11 = 00 00 08 f5  
x11  
x12 = 08

Answer	x12
A	0x8
B	0xf8
C	0x3
D	0xffffffff8

What's the final value in x12?

# 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	0xffffffff8

# Two Reasons for The Answer...

- The immediate got sign extended...
  - So `0xffffffff8f5` 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 `0xfffffffff8`
- 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
Bitwise AND	&	&	<b>and</b>
Bitwise OR			<b>or</b>
Bitwise XOR	^	^	<b>xor</b>
Shift left logical	<<	<<	<b>sll</b>
Shift right	>>	>>	<b>srl/sra</b>



# Logical Shifting

- Shift Left Logical: `slli x11, x12, 2` #  $x11 = x12 \ll 2$ 
  - 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`<sub>16</sub>

`0000 0000 0000 0000 0000 0000 0000 0010`<sub>2</sub>

After: `0000 0008`<sub>16</sub>

`0000 0000 0000 0000 0000 0000 0000 1000`<sub>2</sub>

What arithmetic effect does shift left have?

- Shift Right Logical: `srlr` 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 (inserting sign bit into empty bits)
- For example, if register x10 contained  
 $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{\text{two}} = -25_{\text{ten}}$
- If execute `sra x10, x10, 4`, result is:  
 $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$
- 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

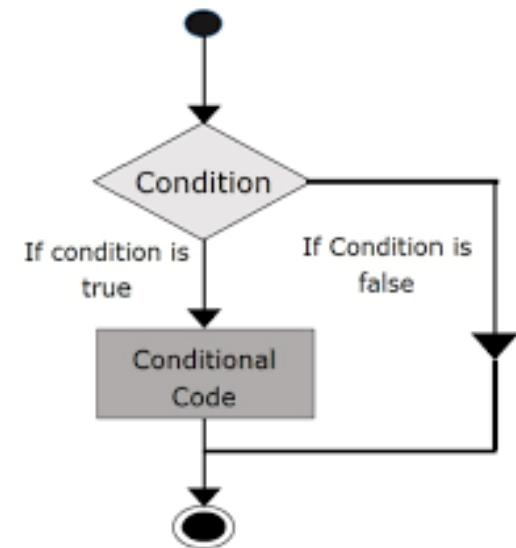
```
int A[100]; /* x13 */
int i;      /* x14 */
...
g = h + A[i]; /* h = x12, g = x11, tmp = x15 */
```

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

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

# Decision Making / Control Flow Instructions

- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for *if-else-statements* and *looping* in standard programming languages
- 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 instructions for this call *jumps*

# 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
  - At what address in the code is that label ...
- Labels give control flow instructions, such as jumps and branches, a place to go ...
  - e.g. `bne x0 x2 foo`
- The assembler in outputting the code does the necessary calculation so the jump or branch will go to the right place

# Example *if* Statement

- Assuming assignments below, compile *if* block

$f \rightarrow \mathbf{x10}$     $g \rightarrow \mathbf{x11}$     $h \rightarrow \mathbf{x12}$

$i \rightarrow \mathbf{x13}$     $j \rightarrow \mathbf{x14}$

**if** (**i == j**)

**f = g + h;**

**bne x13,x14,done**

**add x10,x11,x12**

**done :**



# Example *if-else* Statement

- Assuming assignments below, compile

$f \rightarrow \mathbf{x10}$      $g \rightarrow \mathbf{x11}$      $h \rightarrow \mathbf{x12}$      $i \rightarrow \mathbf{x13}$      $j \rightarrow \mathbf{x14}$

<code>if (i == j)</code>	<code>bne x13,x14,else</code>
<code>    f = g + h;</code>	<code>add x10,x11,x12</code>
<code>else</code>	<code>j done #jump</code>
<code>    f = g - h;</code>	<code>else: sub x10,x11,x12</code>
	<code>done:</code>

# Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C); General programs need to test <, >, >=, <= as well.

“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 its unsigned version) also exist:  
bge, bgeu

# But RISC philosophy...

- RISC-V doesn't have "branch if greater than" or "branch if less than or equal"
- Instead you can reverse the arguments:

$$A > B \quad \equiv \quad B < A$$

$$A \leq B \quad \equiv \quad B \geq A$$

- The assembler defines pseudo-instructions for your convenience:
  - `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];
```

Loop has 7 instructions

```
# 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   # A + i  
lw x13, 0(x13)     # *(A + i)  
add x10, x10, x13  # increment sum  
addi x11, x11, 1   # i++  
j Loop             # Iterate  
exit:
```

# C Loop Mapped to RISC-V Assembly

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

Loop now 6 instructions

Slightly optimized

```
# Assume x8 holds base address of A
# Assign x10=sum, x11=i*4
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x12,x0,80  # x12=20*4
Loop:
bge x11, x12, exit:
add x13, x11, x8 # A + i
lw x13, 0(x13)   # *(A + i)
add x10, x10, x13 # increment sum
addi x11, x11, 4  # i++
j Loop           # Iterate
exit:
```

# More optimizations:

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

- Inner loop is now 4 instructions rather than 6
  - Directly increment ptr into A array
  - 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

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

# Outline

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

# In Conclusion,...

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