CS 61C:
Great Ideas in Computer Architecture
Introduction to Assembly Language and RISC-V Instruction Set Architecture

Instructors:
Nick Weaver & John Wawrzynek
http://inst.eecs.Berkeley.edu/~cs61c

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

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

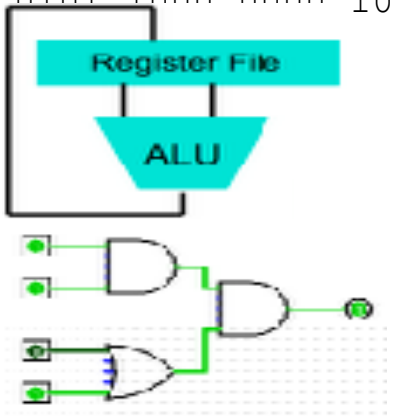
Logic Circuit Description (Circuit Schematic Diagrams)

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

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

```
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
```

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



Instruction Set Architecture (ISA)

- Job of a CPU (*Central Processing Unit*, aka *Core*): execute *instructions*
- Instructions: CPU's primitives 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 change the sequence of instruction.
- 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), Intel IA64, ...

Instruction Set Architectures

- Early trend: add more instructions to new CPUs for elaborate operations
  - Made assembly language programming easier.
  - VAX architecture had an instruction to compute polynomials!

```
result = C[0]+x**0 + x*(C[1] + x*(C[2] + ... x*C[d]))
```

- RISC philosophy (Cocke IBM, Patterson UCB, Hennessy Stanford, 1980s) – *Reduced Instruction Set Computing*
  - Keep the instruction set small and simple, in order to build fast hardware
  - Let compiler generate software do complicated operations by composing simpler ones

Assembly Language Programming

ARM

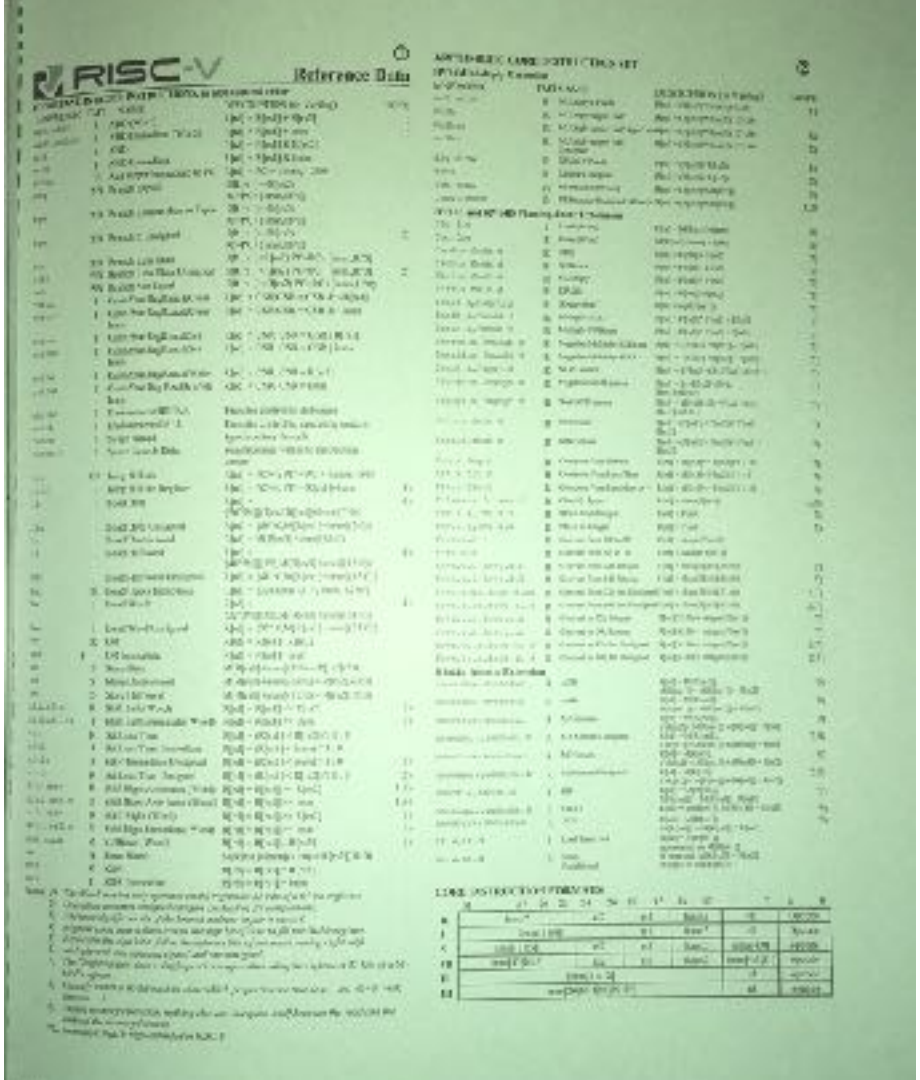
```
LDR r0,[p_a]
LDR r1,[p_b]
ADD r3,r0,r1
STR r3,[p_w]
LDR r2,[p_c]
ADD r0,r2,r3
STR r0,[p_x]
LDR r0,[p_d]
ADD r3,r2,r0
STR r3,[p_y]
```

x86

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

- 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 where complex and compilers where 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
  1. Best way to understand what compilers do to generate machine code
  2. Best way to understand what the CPU hardware does
  3. *Plus its great fun!*

RISC-V
Green Card
(in textbook)



http://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Green_Sheet.pdf

Inspired by the IBM 360 “Green Card”

IBM System/360 Reference Data

MACHINE INSTRUCTIONS

NAME	MNEMONIC	OP CODE	FOR MAT	OPERANDS
Add (c)	AR	1A	RR	R1,R2
Add (c)	A	5A	RX	R1,D2(X2,B2)
Add Decimal (c,d)	AP	FA	SS	D1(L1,B1),D2(L2,B2)
Add Halfword (c)	AH	4A	RX	R1,D2(X2,B2)
Add Logical (c)	ALR	1E	RR	R1,R2
Add Logical (c)	AL	5E	RX	R1,D2(X2,B2)
AND (c)	NR	14	RR	R1,R2
AND (c)	N	54	RX	R1,D2(X2,B2)
AND (c)	NI	94	SI	D1(B1),I2
AND (c)	NC	D4	SS	D1(L,B1),D2(B2)
Branch and Link	BALR	05	RR	R1,R2
Branch and Link	BAL	45	RX	R1,D2(X2,B2)
Branch and Store (a)	BASR	0D	RR	R1,R2
Branch and Store (a)	BAS	4D	RX	R1,D2(X2,B2)
Branch on Condition	BCR	07	RR	M1,R2
Branch on Condition	BC	47	RX	M1,D2(X2,B2)
Branch on Count	BCTR	06	RR	R1,R2
Branch on Count	BCT	46	RX	R1,D2(X2,B2)
Branch on Index High	BXH	86	RS	R1,R3,D2(B2)
Branch on Index Low or Equal	BXLE	87	RS	R1,R3,D2(B2)
Compare (c)	CR	19	RR	R1,R2
Compare (c)	C	59	RX	R1,D2(X2,B2)
Compare Decimal (c,d)	CP	F9	SS	D1(L1,B1),D2(L2,B2)
Compare Halfword (c)	CH	49	RX	R1,D2(X2,B2)
Compare Logical (c)	CLR	15	RR	R1,R2
Compare Logical (c)	CL	55	RX	R1,D2(X2,B2)
Compare Logical (c)	CLC	D5	SS	D1(L,B1),D2(B2)
Compare Logical (c)	CLI	95	SI	D1(B1),I2
Convert to Binary	CVB	4F	RX	R1,D2(X2,B2)
Convert to Decimal	CVD	4E	RX	R1,D2(X2,B2)
Diagnose (p)		83	SI	
Divide	DR	1D	RR	R1,R2
Divide	D	5D	RX	R1,D2(X2,B2)
Divide Decimal (d)	DP	FD	SS	D1(L1,B1),D2(L2,B2)
Edit (c,d)	ED	DE	SS	D1(L,B1),D2(B2)
Edit and Mark (c,d)	EDMK	DF	SS	D1(L,B1),D2(B2)
Exclusive OR (c)	XR	17	RR	R1,R2
Exclusive OR (c)	X	57	RX	R1,D2(X2,B2)
Exclusive OR (c)	XI	97	SI	D1(B1),I2
Exclusive OR (c)	XC	D7	SS	D1(L,B1),D2(B2)
Execute	EX	44	RX	R1,D2(X2,B2)
Halt I/O (c,p)	HIO	9E	SI	D1(B1)
Insert Character	IC	43	RX	R1,D2(X2,B2)
Insert Storage Key (a,p)	ISK	09	RR	R1,R2
Load	LR	18	RR	R1,R2
Load	L	58	RX	R1,D2(X2,B2)
Load Address	LA	41	RX	R1,D2(X2,B2)
Load and Test (c)	LTR	12	RR	R1,R2
Load Complement (c)	LCR	13	RR	R1,R2
Load Halfword	LH	48	RX	R1,D2(X2,B2)
Load Multiple	LM	98	RS	R1,R3,D2(B2)
Load Multiple Control (a,p)	LMC	B8	RS	R1,R3,D2(B2)
Load Negative (c)	LNR	11	RR	R1,R2
Load Positive (c)	LPR	10	RR	R1,R2
Load PSW (c,p)	LPSW	82	SI	D1(B1)
Load Real Address (c,e,p)	LRA	B1	RX	R1,D2(X2,B2)
Move	MVI	92	SI	D1(B1),I2
Move	MVC	D2	SS	D1(L,B1),D2(B2)
Move Numerics	MVN	D1	SS	D1(L,B1),D2(B2)
Move with Offset	MVO	F1	SS	D1(L1,B1),D2(L2,B2)
Move Zones	MVZ	D3	SS	D1(L,B1),D2(B2)
Multiply	MR	1C	RR	R1,R2
Multiply	M	5C	RX	R1,D2(X2,B2)
Multiply Decimal (d)	MP	FC	SS	D1(L1,B1),D2(L2,B2)
Multiply Halfword	MH	4C	RX	R1,D2(X2,B2)
OR (c)	OR	16	RR	R1,R2
OR (c)	O	56	RX	R1,D2(X2,B2)
OR (c)	OI	96	SI	D1(B1),I2

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
- Experiencing rapid uptake in both industry and academia
- Supported by growing shared software ecosystem
- 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

Foundation Members (60+)

Platinum:

Gold, Silver, Auditors:

Rumble

Development

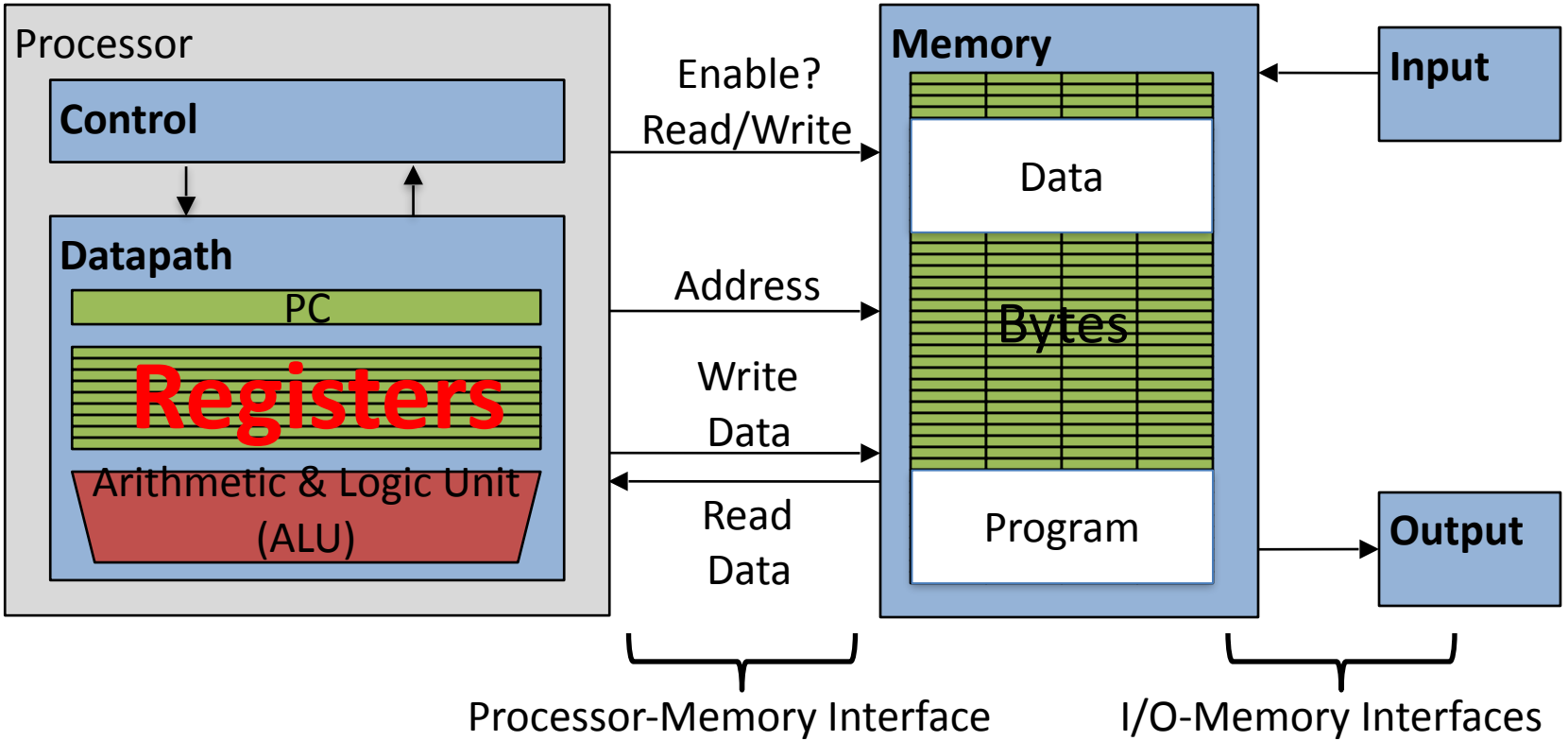
Outline

- 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, 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
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast to access (faster than 1 ns - light travels 1 foot in 1 ns!!!)

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 8 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, described later
  - Why 32? Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- 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 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)
- In Assembly Language:
  - Registers have no type;
  - Operation determines how register contents are interpreted

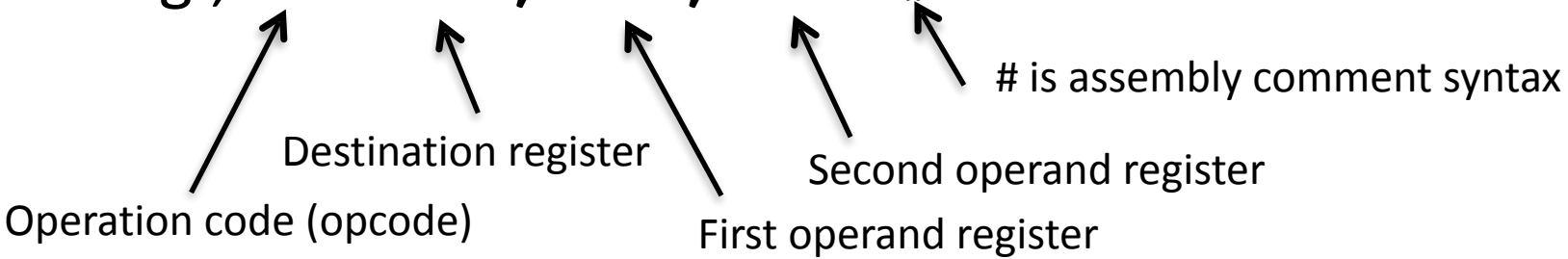
Outline

- 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

Addition and Subtraction of Integers Example 1

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

```
add x10, x1, x2  # temp = b + c
add x10, x10, x3 # temp = temp + d
sub x10, x10, x4 # 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:

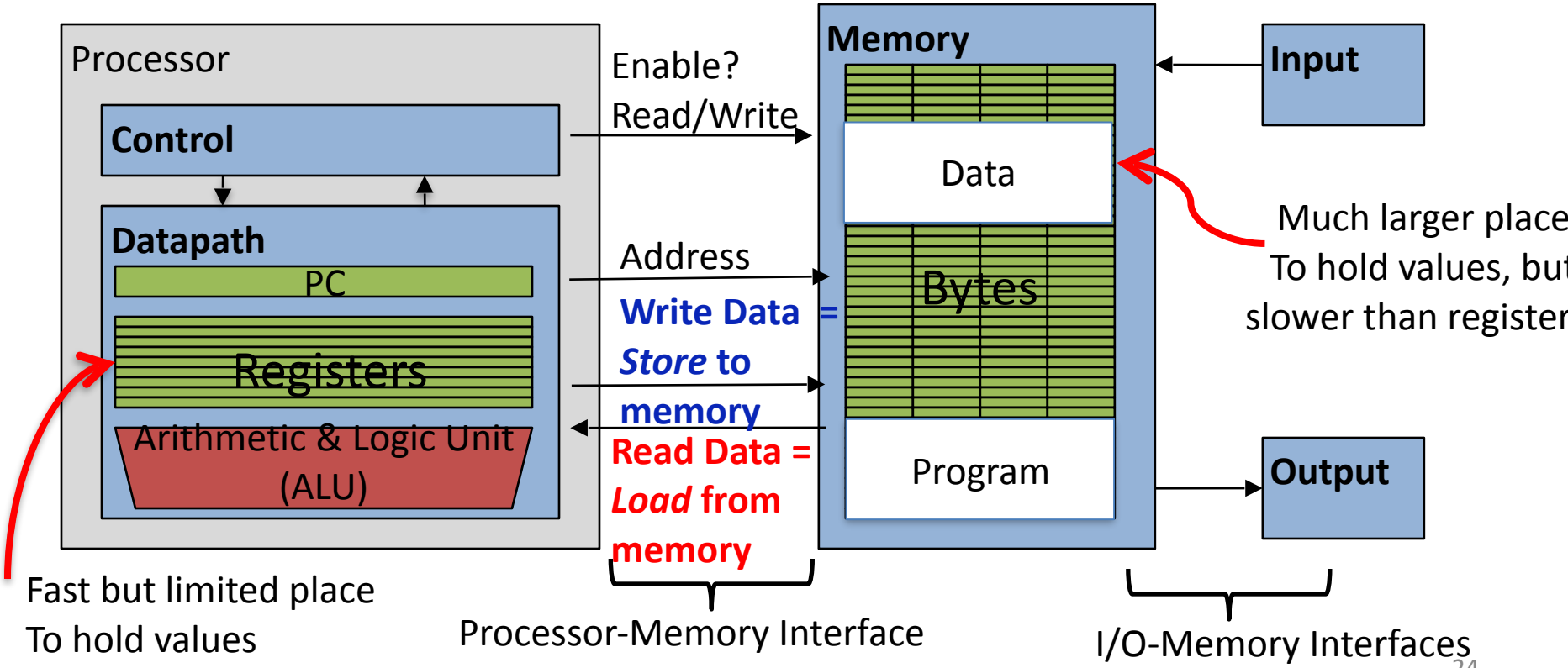
```
addi x3,x4,-10   (in RISC-V)
f = g - 10       (in C)
```

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

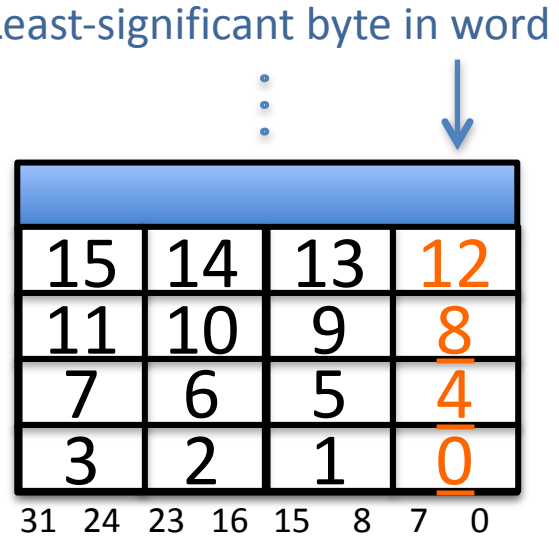
```
addi x3,x4,0     (in RISC-V) same as
f = g            (in C)
```

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)–works fine if 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)



Words in memory must start at byte addresses that are even multiples of 4, i.e., words must be aligned.
Note: aligned words have the low 2 bits of their address = 0.

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)

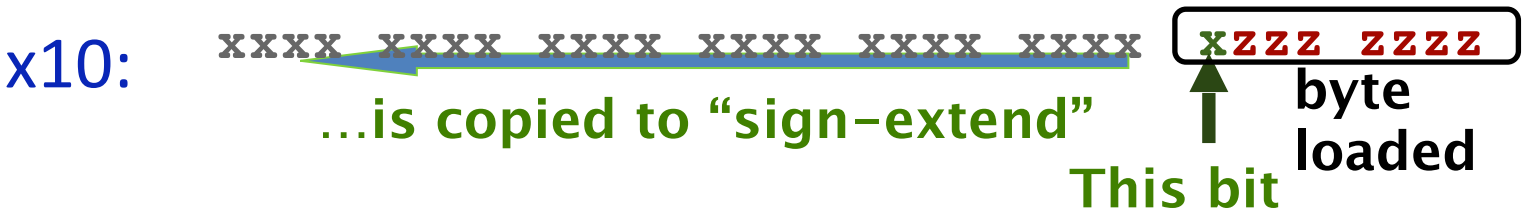
Note: `12,40` – offsets in bytes

x13+12 and x13+40 must be multiples of 4

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`**?



Your turn - clickers

```
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

Your turn - clickers

```
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

Administrivia

- The Project 1 deadline extended to Thursday, 11:59pm!
- Send DSP letters to Pejie <li_paige@berkeley.edu>.
- There will be a guerrilla section Thursday 7-9PM.
- Two weeks to Midterm #1!
- Project 2-1 release later this week or early next, due 2/16.
- Project 2-2 release right after midterm and due 2/23.

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 logical	>>	>>	`srl`

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: $0000\ 0008_{hex}$

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$

  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

Computer Decision Making

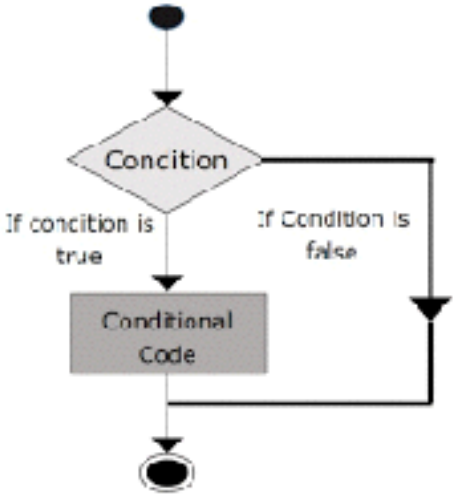
- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for programming languages: *if*-statement

- 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`)*

Outline

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

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,skip
  f = g + h;               add x10,x11,x12
        .            skip:          .
        .                           .
        .                           .
```

Example *if-else* Statement

- Assuming assignments below, compile

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

```
if (i == j)
  f = g + h;
else
  f = g - h;
```

```
      bne x13,x14,else
      add x10,x11,x12
      j done
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) // treat registers as signed integers
  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.

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 x9=A, x10=sum, x11=i
add x9, x8, x0   # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x13,x0,20   # x13=20
Loop:
lw x12, 0(x9)    # x12=A[i]
add x10,x10,x12 # sum+=
addi x9,x9,4     # &A[i++]
addi x11,x11,1   # i++
blt x11,x13,Loop
```

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*