CS 110
Computer Architecture
Lecture 5:
Intro to Assembly Language, RISC-V Intro

Instructor:

Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

History

50 years ago: Apollo Guidance Computer programmed in Assembly

30x30x30cm, 32 kg.
10,000 lines of machine code manually entered – tons of easter eggs!

abcnews.go.com/Technology/apollo-11s-source-code-tons-easter-eggs-including/story?id=40515222

Margaret Hamilton with the code she wrote.

- Lead Apollo flight software designer.
- Came up with the idea of naming the discipline, "software engineering"
- https://en.wikipedia.org/wiki/Margaret_Hamilton_%28scientist%29

```
        TC      BANKCALL        # TEMPORARY, I HOPE HOPE HOPE
        CADR    STOPRATE        # TEMPORARY, I HOPE HOPE HOPE
        TC      DOWNFLAG        # PERMIT X-AXIS OVERRIDE
```

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

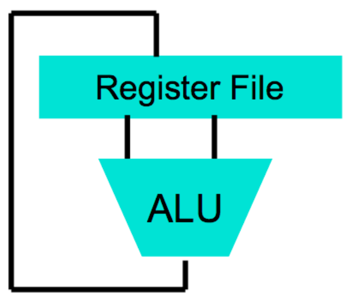
Logic Circuit Description (Circuit Schematic Diagrams)

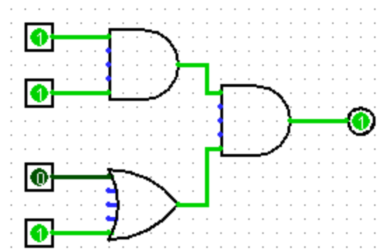
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    xt0, 0(x2)
lw    xt1, 4(x2)
sw    xt1, 0(x2)
sw    xt0, 4(x2)
```

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
```





Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Intel IA64, ...

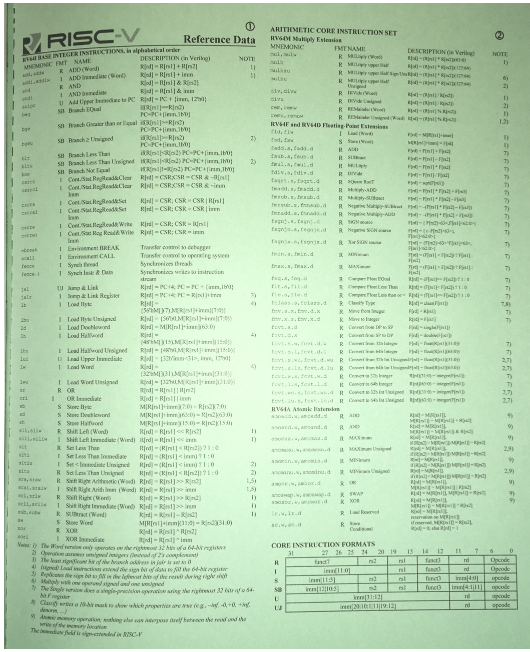
Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!

- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) –
  Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.

RISC-V Architecture

- New open-source, license-free ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)



RISC-V Green Card

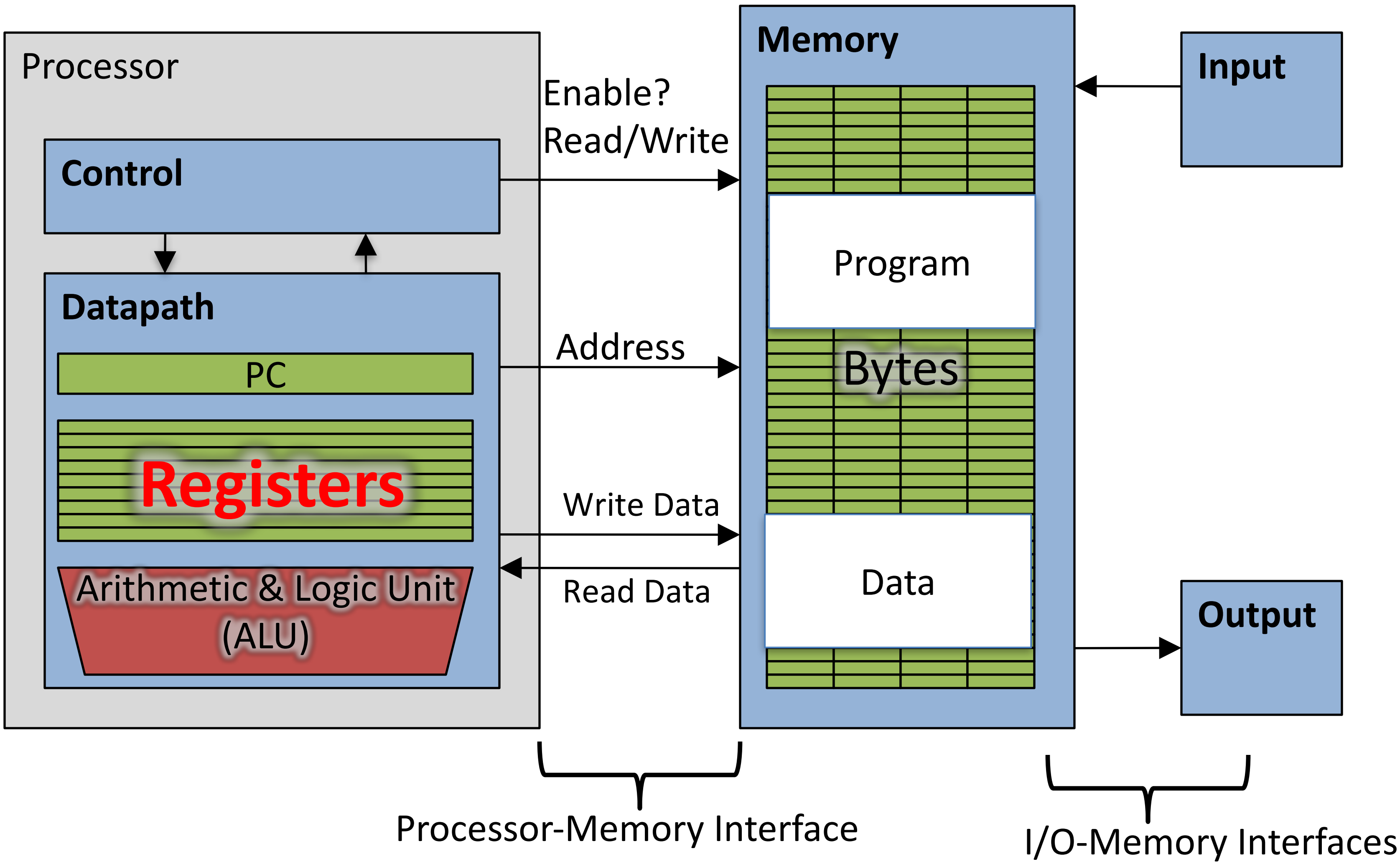
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
  - RISC-V has exponential adoption rate

Assembly Variables: Registers

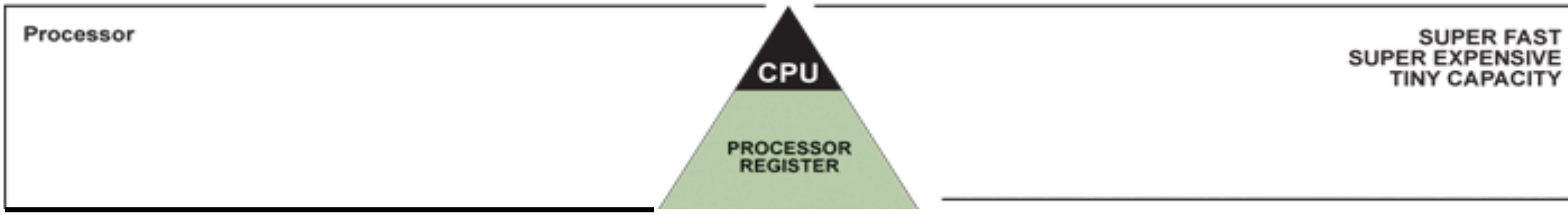
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!

- Benefit: Since registers are directly in hardware, they are very fast
(faster than 1 ns - light travels 30cm in 1 ns!!!)

Registers, inside the Processor



Great Idea #3: Principle of Locality / Memory Hierarchy



Number of Registers

- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: Assembly code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V
  - Why 32? Smaller is faster, but too small is bad.
- Each RISC-V register is 32 bits wide (in RV32 variant)
  - Groups of 32 bits called a word in RV32
  - P&H textbook uses 64-bit variant RV64 (doubleword)

RISC-V Registers

- Registers are numbered from 0 to 31
- Number references:
  - x0, x1, x2, … x30, x31
- x0 : special: always holds value zero
  => only 31 registers to hold variable values
- Each register can be referred to by number or name
  - Cover names later

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first 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 as (cannot mix and match *int* and *char* variables).
- In Assembly Language, registers have no type; operation determines how register contents are treated

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for RISC-V comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    /* comment */
    so they can span many lines

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:
  - One two, three, four `add x1, x2, x3`
  - where:
  - One = operation by name
  - two = operand getting result ("destination")
  - three = 1st operand for operation ("source1")
  - four = 2nd operand for operation ("source2")
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- 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 (3/4)

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

```
add x10, x1, x2   # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

Addition and Subtraction of Integers (4/4)

- How do we do this?

  f = (g + h) - (i + j);

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h
add x6, x22, x23 # b_temp = i + j
sub x19, x5, x6  # f = (g + h)- (i + j)
```

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- 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.

Immediates

- There is no Subtract Immediate in RISC-V: Why?
  - There are add and sub, but no addi counterpart
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - addi …, -X = subi …, X => so no subi

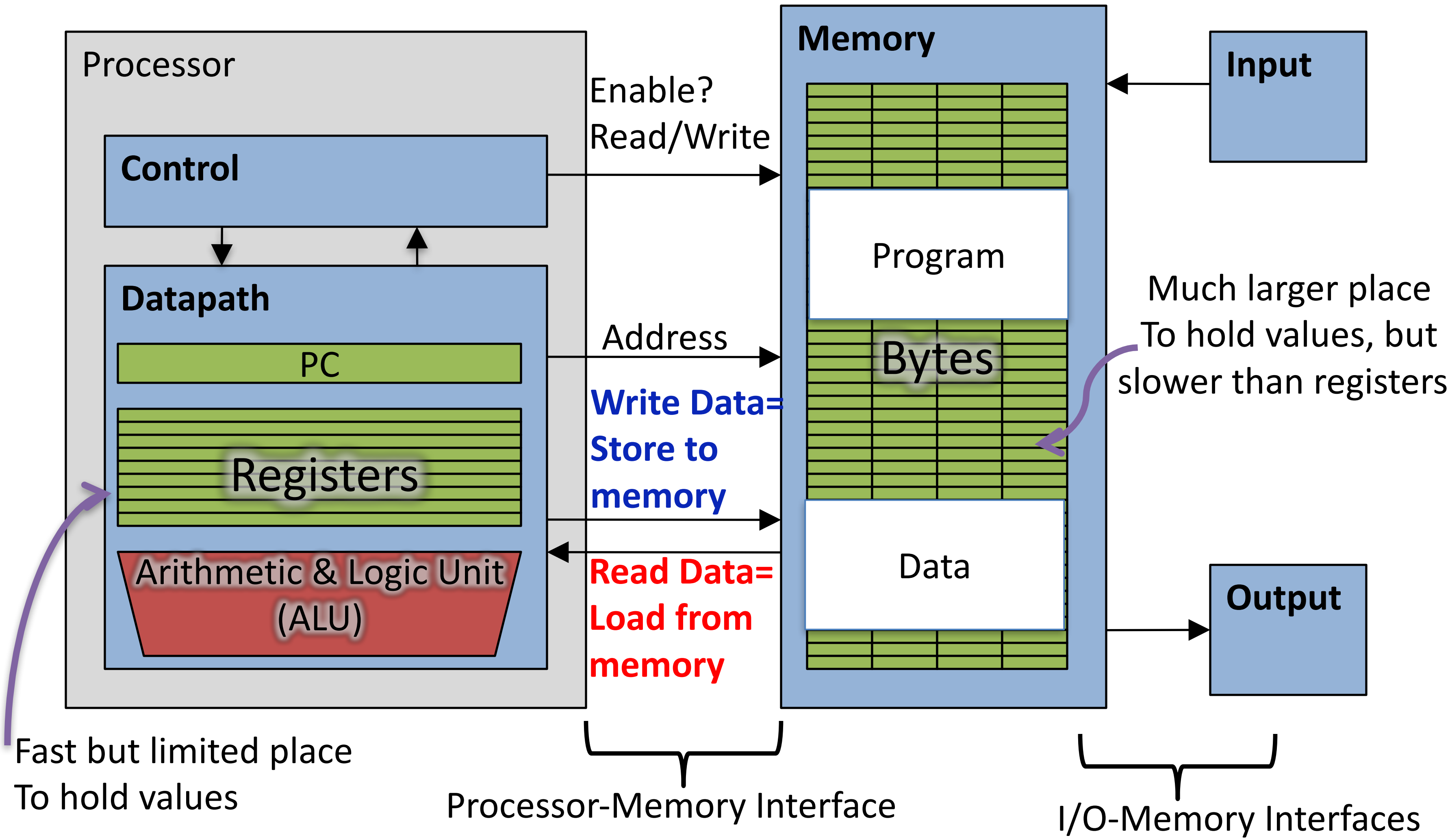
    `addi x3,x4,-10` (in RISC-V)

    f = g - 10 (in C)
  - where RISC-V registers `x3, x4` are associated with C variables f, g

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (`x0`) is 'hard-wired' to value 0; e.g.
  - `add x3,x4,x0` (in RISC-V)
  - f = g (in C)
  - where RISC-V registers `x3,x4` are associated with C variables f, g
- Defined in hardware, so an instruction
  - `add x0,x3,x4` will not do anything!

Data Transfer: Load from and Store to memory



Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – 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 leftmost byte – most significant byte (i.e. Big-endian convention)

Little Endian:
Most significant byte in a word

…	…	…	…
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Big Endian:
Most significant byte in a word

en.wikipedia.org/wiki/Big_endian

Big Endian vs. Little Endian

Big-endian and little-endian from Jonathan Swift's *Gulliver's Travels*

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3 BYTE2 BYTE1 BYTE0

00000000 00000000 00000100 00000001

Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

Examples

Names in China (e.g., Schwertfeger, Sören)

Java Packages: (e.g., org.mypackage.HelloWorld)

Dates done correctly ISO 8601 YYYY-MM-DD (e.g., 2018-09-07)

Eating Pizza crust first

Unix file structure (e.g., /usr/local/bin/python)

"Network Byte Order": most network protocols

IBM z/Architecture; very old Macs

Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Examples

Names in the west (e.g., Sören Schwertfeger)

Internet names (e.g., sist.shanghaitech.edu.cn)

Dates written in England DD/MM/YYYY (e.g., 07/09/2018)

Eating Pizza skinny part first (the normal way)

CANopen

Intel x86; RISC-V

bi-endian: ARM (runs mostly little endian), MIPS, IA-64, PowerPC

RISC-V: Little Endian

(E.g., 0xC2=0b 1100 0010)

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Little Endian
Most significant byte in a word
(numbers are addresses)

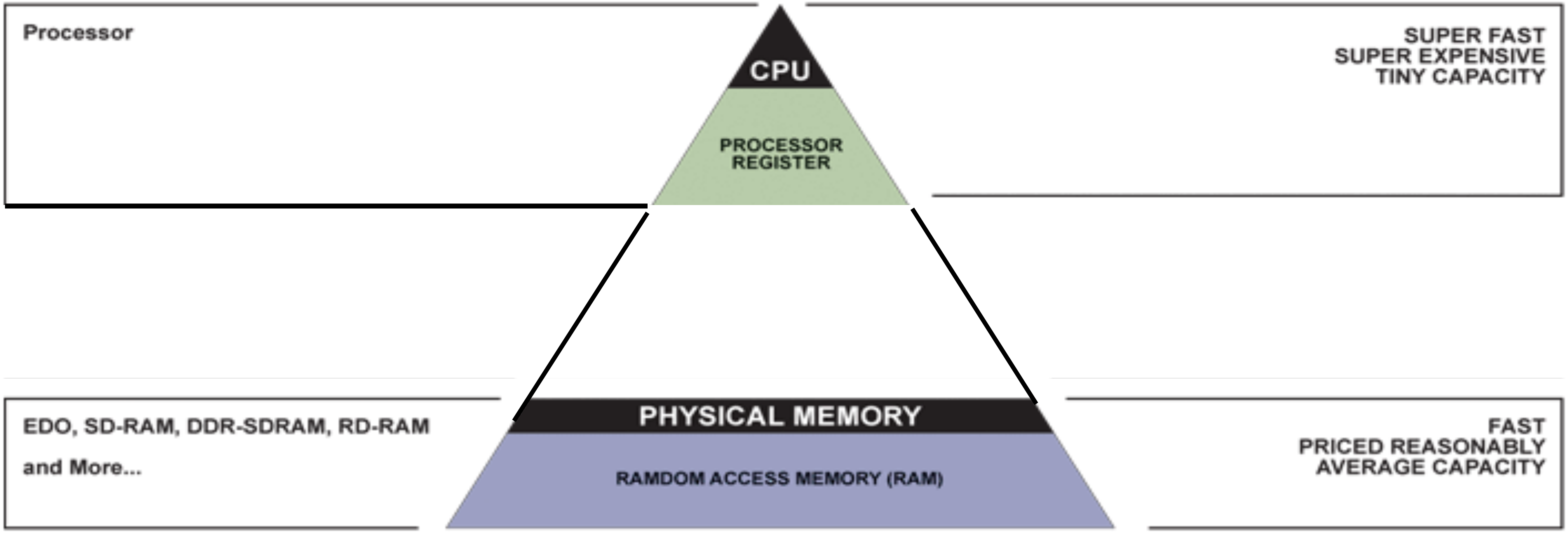
...	...	...	...
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

- Hexadecimal number:
  0xFD34AB88 ($4{,}248{,}087{,}432_{ten}$) =>
  - Byte 0: 0x88 (136_{ten})
  - Byte 1: 0xAB (171_{ten})
  - Byte 2: 0x34 (52_{ten})
  - Byte 3: 0xFD (253_{ten})

Address:	64 address of word (e.g. int)			
Address:	64	65	66	67
Data:	0x88	0xAB	0x34	0xFD

- Little Endian: The "Endianess" is little:
  - It starts with the smallest (least significant) Byte
  - Swapped from how we write the number

Great Idea #3: Principle of Locality / Memory Hierarchy



Speed of Registers vs. Memory

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

Load from Memory to Register

- C code

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

Data flow

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

```
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

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

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

```
lw   x10,12(x15)     # Temp reg x10 gets A[3]
add  x10,x12,x10     # Temp reg x10 gets h + A[3]
sw   x10,40(x15)     # A[10] = h + A[3]
```

Data flow

Note: `x15` – base register (pointer)

`12,40` – offsets in bytes

~~x15+12 and x15+40 must be multiples of 4~~

Question:

We want to translate `*x = *y +1` into RISC-V
(x, y **int** pointers stored in: `x10 x11`)

```
A:    addi x10,x11,1

B:    lw   x10,1(x11)
      sw   x11,0(x10)

C:    lw   x13,0(x11)
      addi x13,x13,1
      sw   x13,0(x10)

D:    sw   x13,0(x11)
      addi x13,x13,1
      lw   x13,0(x10)

E:    lw   x10,1(x13)
      sw   x11,0(x13)
```

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`**?

x10:

xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz

…is copied to “sign-extend”

This bit

byte loaded

Question. What's in `x12`?

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

A:	0x0
B:	0x3
C:	0x5
D:	0xF
E:	0xFFFFFFFF

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`**
Bit-by-bit NOT	~	~	**`xori`**
Shift left	<<	<<	**`sll`**
Shift right	>>	>>	**`srl`**

Logic Shifting

- Shift Left: `sll 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?

multiply with 2^n

- Shift Right: `srl` is opposite shift; >>

Arithmetic Shifting

- Shift right arithmetic 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 executed 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

Peer Instruction

We want to translate `*x = *y` into RISC-V

x, y ptrs stored in: `x3` `x5`

```
1: add x3,    x5, zero
2: add x5,    x3, zero
3: lw  x3, 0(x5)
4: lw  x5, 0(x3)
5: lw  x8, 0(x5)
6: sw  x8, 0(x3)
7: lw  x5, 0(x8)
8: sw  x3, 0(x8)
```

A	1
B	2
C	3
D	4
E	5→6
F	6→5
G	7→8

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement

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

  `beq register1,register2,L1`

  means: go to statement labeled L1
  if (value in register1) == (value in register2)
  ....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`**)

- **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

  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 Exit
  f = g - h;     Else:     sub x10,x11,x12
                 Exit:
```

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;

C Loop Mapped to RISC-V Assembly

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

"And in Conclusion…"

- In RISC-V Assembly Language:
  - Registers replace C variables
  - One instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- In RV32, words are 32b
- Instructions:

  `add`, `addi`, `sub`
- Registers:
  - 32 registers, referred to as `x0` – `x31`
  - Zero: `x0`

“And in Conclusion…”

- Memory is byte-addressable, but lw and sw access one word at a time.
- A pointer (used by lw and sw) is just a memory address, we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.
- RISC-V Decision making instructions are the conditional branches: `beq` and `bne`.
- Instructions:
  - **`lw, sw, lb, sb, lbu, beq, bne, blt, bltu, bge, j`**