

CS 110

Computer Architecture

Lecture 4: *Intro to Assembly Language, RISC-V Intro*

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C



Admin

- Life lecture always covers the 1st video of next lecture
- All videos of next lecture will be published at the day of this (so the previous) lecture
- 1st video is not required to watch, if you understood this live lecture
 - there are NO mandatory piazza polls in the 1st video
- Other videos are MANDATORY to watch
 - You are REQUIRED to do the according polls till the day of that lecture (day where the video is published on the website).



Admin

- Labs 2-9 today
- Prepare checkoffs!
- When meeting with TA, you should be able to do all checkoffs.
- On campus: go to lab, do the lab, at the end of the class, checkoff with TA...
- Prepare zoom, practice screen sharing (whole screen)



Admin

- HW 2 is due soon (March 12)
- HW 3 and Project 1 will be released soon!

History

51 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

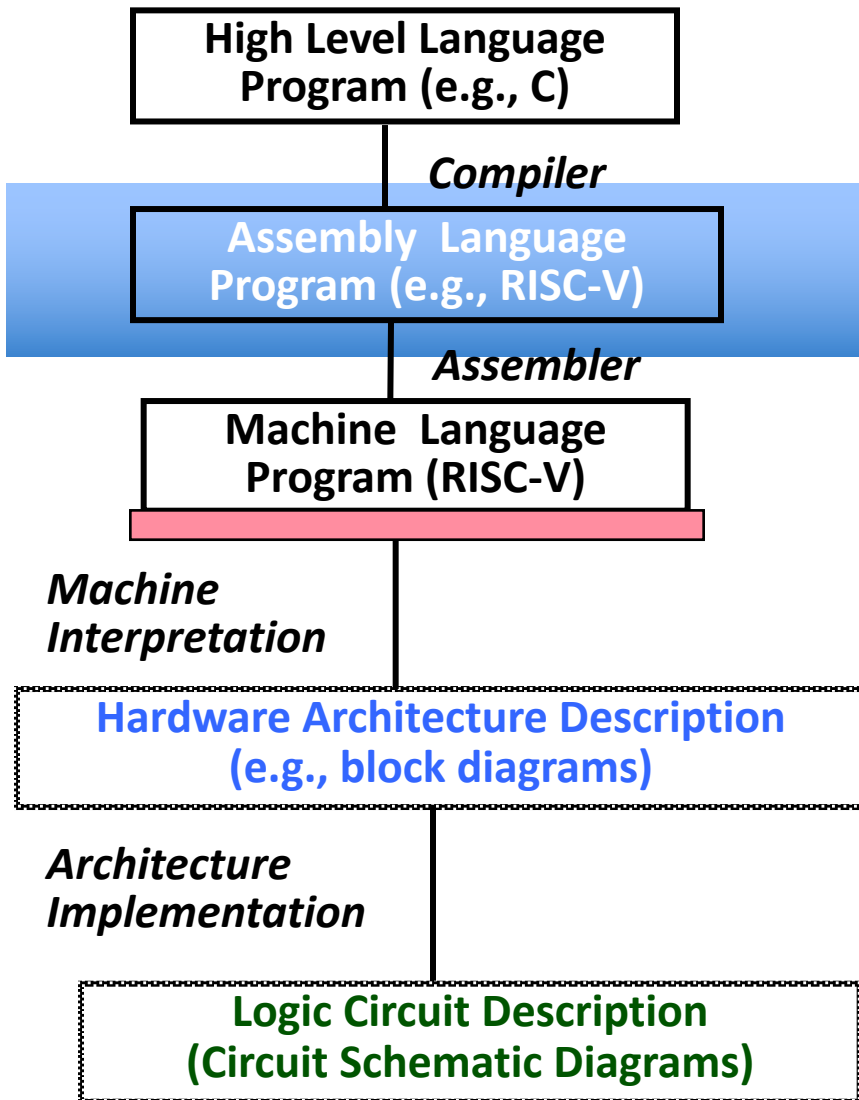


Margaret Hamilton with the code she wrote.

- 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

179	TC	BANKCALL	# TEMPORARY, I HOPE HOPE HOPE
180	CADR	STOPRATE	# TEMPORARY, I HOPE HOPE HOPE
181	TC	DOWNFLAG	# PERMIT X-AXIS OVERRIDE

Levels of Representation/Interpretation

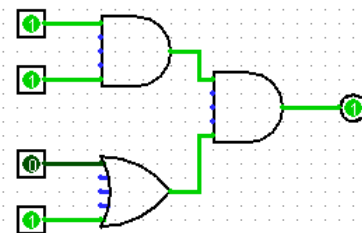
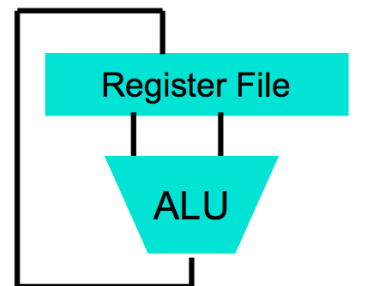


```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
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, ...

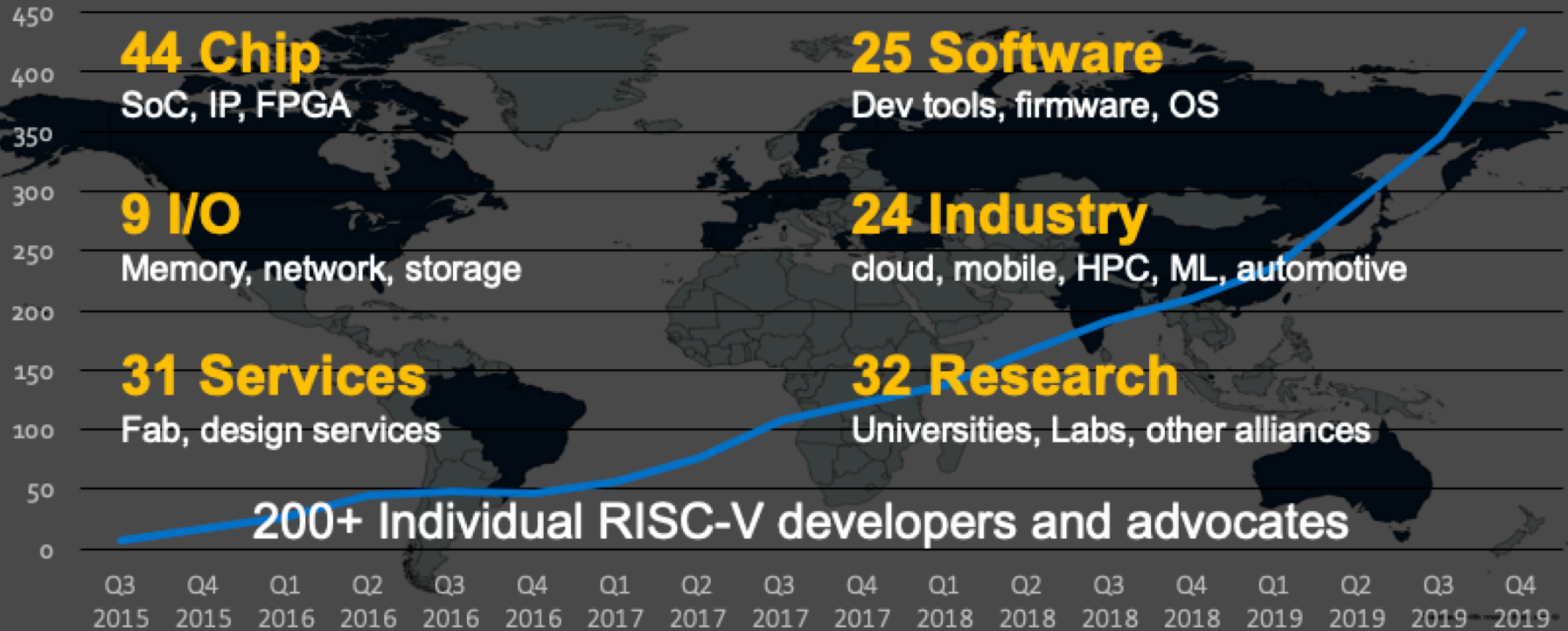
Instruction Set Architectures

- 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.

- New open-source, license-free RISC 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 standard maintained by non-profit RISC-V Foundation

More than 435 RISC-V Members

across 33 Countries Around the World



Platinum Foundation Members



Alibaba Group
PLATINUM



Andes Technology
FOUNDING PLATINUM



Antmicro
FOUNDING PLATINUM



Berkeley Architecture Research
FOUNDING PLATINUM



Bluespec
FOUNDING PLATINUM



ConnectFree
PLATINUM



Cortus
FOUNDING PLATINUM



Google
FOUNDING PLATINUM



Huami
PLATINUM



Microchip Technology
FOUNDING PLATINUM



Micron Technology
PLATINUM



NVIDIA
FOUNDING PLATINUM



NXP
PLATINUM



Orion
PLATINUM



Qualcomm
FOUNDING PLATINUM



Rambus Inc.
FOUNDING PLATINUM



Samsung
PLATINUM



Sanechips Technology Co.
PLATINUM



SiFive
FOUNDING PLATINUM



Thales
PLATINUM



Western Digital
FOUNDING PLATINUM

RISC-V in China

- 33 Chinese members in the global RISC-V Foundation
- 500 attendees at the China RISC-V Forum in Nov 2019
- RISC-V International Open Source Laboratory (RIOS Laboratory) research at Tsinghua-Berkeley Shenzhen Institute (TBSI) June 2019
- Alibaba processor achieves 7.1 Coremark/MHz at a frequency of 2.5GHz on a 12nm process node, which is 40 percent more powerful than any RISC-V processor produced to date. – EE/Times July 2019
- GigaDevice launched world's first general-purpose microcontroller based on RISC-V for the IOT market. – EE Times 26 Aug 2019
- Huami's upcoming Huangshan 1S Processor in 7nm
Huami one of the top wearable manufacturers; August 27, 2019

Why RISC-V in CS110?

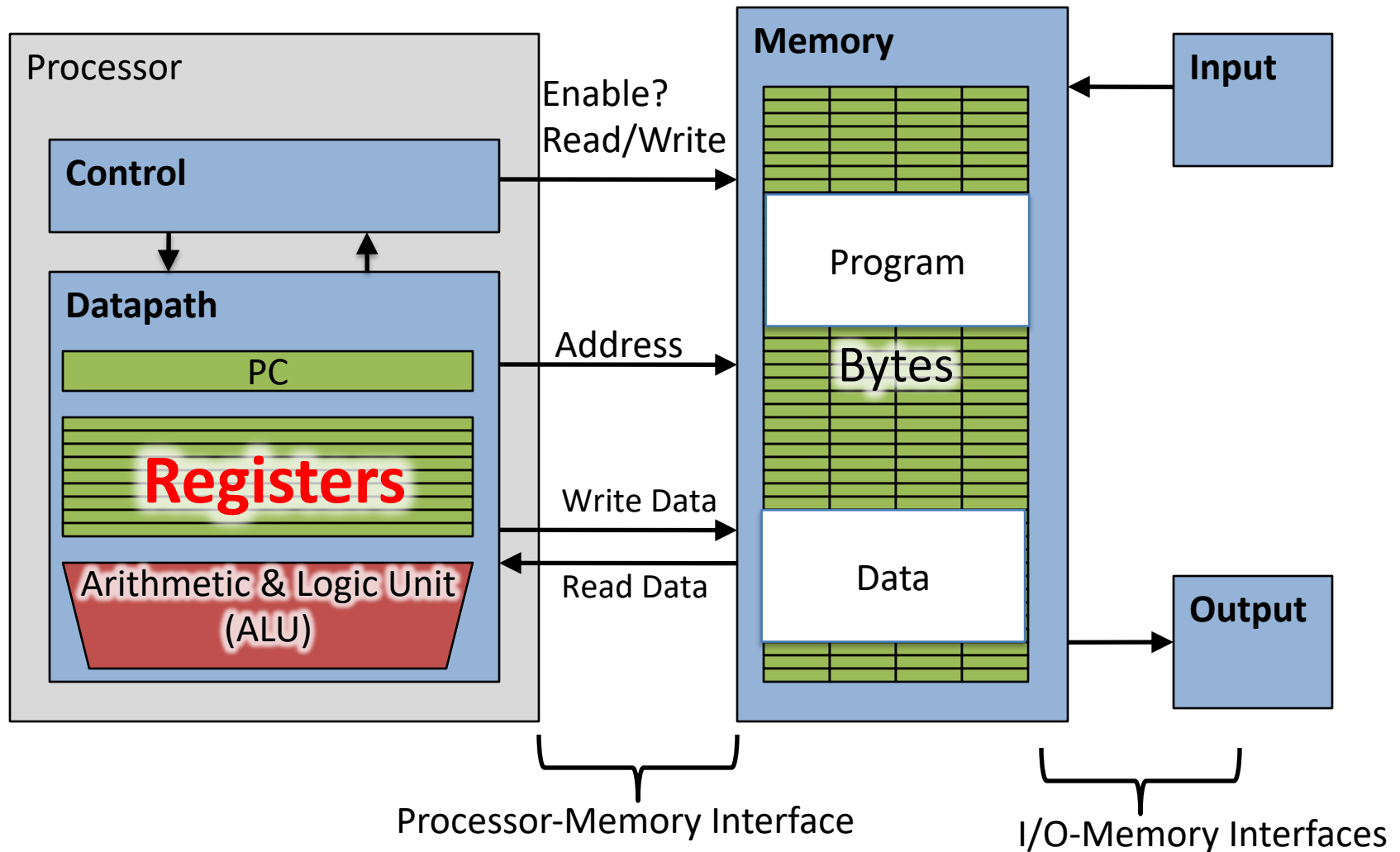
- Why RISC-V instead of Intel 80x86?
 - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
- It is a very very clean RISC
 - No real additional "optimizations"
- Generally only one way to do any particular thing
 - Only exception is two different atomic operation options: Load Reserved/Store Conditional Atomic swap/add/etc...

The image shows a page from the RISC-V Reference Data document. It contains a table of instructions and their descriptions, organized into sections like 'RISC-V Instruction Set' and 'RISC-V Atomic Instructions'. The table includes columns for instruction names, mnemonics, and descriptions. Below the table, there are sections for 'CORE INSTRUCTION FORMATS' and 'CORE INSTRUCTION FORMATS' showing bit-level details of instructions.

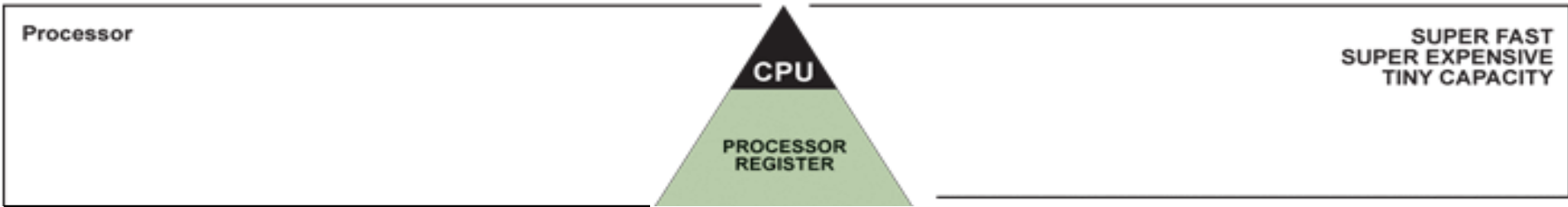
Assembly Variables: Registers

- 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



Live Lecture Instruction:

Put an * in front of your name NOW!

e.g. *_32_ChenHao

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 \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 (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)
```



Q & A



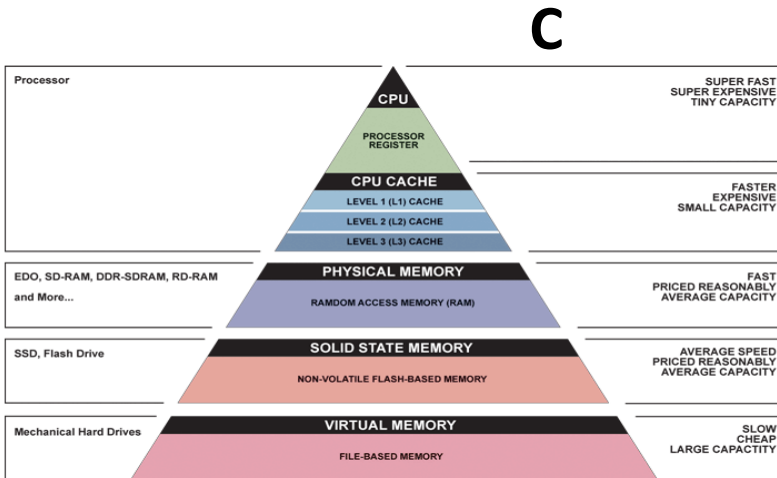
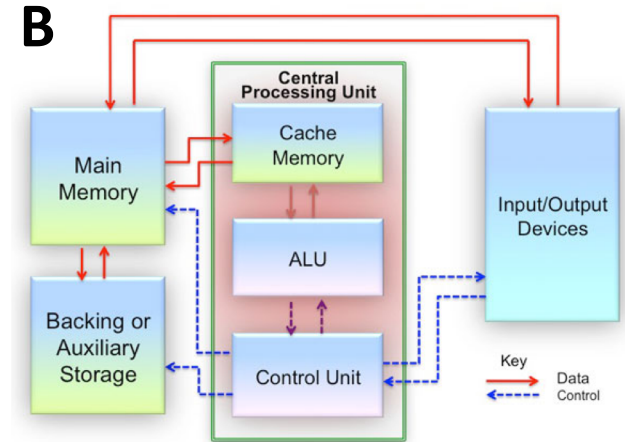
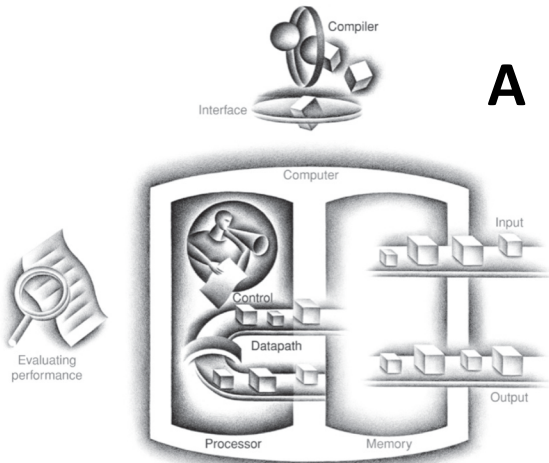
Quiz



Quiz

Piazza: "Online Lecture 3 Quiz"

Overview of the CPU



D RISC-V RV32I / RV64I / RV128I + M, A, F, D, Q, C RISC-V "Green Card"

RISC-V		RV32I (RV32M)		RV64I (RV64M)		RV128I (RV128M)		RISC-V Reference Card	
Instruction	OpCode	OpCode	OpCode	OpCode	OpCode	OpCode	OpCode	OpCode	OpCode
ADDI	0000000	0000000	0000000	0000000	0000000	0000000	0000000	ADDI	0000000
SLTI	0000001	0000001	0000001	0000001	0000001	0000001	0000001	SLTI	0000001
SLTIU	0000010	0000010	0000010	0000010	0000010	0000010	0000010	SLTIU	0000010
ANDI	0000011	0000011	0000011	0000011	0000011	0000011	0000011	ANDI	0000011
ORR	0000012	0000012	0000012	0000012	0000012	0000012	0000012	ORR	0000012
XORI	0000013	0000013	0000013	0000013	0000013	0000013	0000013	XORI	0000013
AND	0000014	0000014	0000014	0000014	0000014	0000014	0000014	AND	0000014
OR	0000015	0000015	0000015	0000015	0000015	0000015	0000015	OR	0000015
XOR	0000016	0000016	0000016	0000016	0000016	0000016	0000016	XOR	0000016
SLL	0000017	0000017	0000017	0000017	0000017	0000017	0000017	SLL	0000017
SRL	0000018	0000018	0000018	0000018	0000018	0000018	0000018	SRL	0000018
SRLLI	0000019	0000019	0000019	0000019	0000019	0000019	0000019	SRLLI	0000019
SHR	0000020	0000020	0000020	0000020	0000020	0000020	0000020	SHR	0000020
SHRLI	0000021	0000021	0000021	0000021	0000021	0000021	0000021	SHRLI	0000021
ANDI	0000022	0000022	0000022	0000022	0000022	0000022	0000022	ANDI	0000022
ORI	0000023	0000023	0000023	0000023	0000023	0000023	0000023	ORI	0000023
XORI	0000024	0000024	0000024	0000024	0000024	0000024	0000024	XORI	0000024
SLLI	0000025	0000025	0000025	0000025	0000025	0000025	0000025	SLLI	0000025
SRLI	0000026	0000026	0000026	0000026	0000026	0000026	0000026	SRLI	0000026
SRLLI	0000027	0000027	0000027	0000027	0000027	0000027	0000027	SRLLI	0000027
SHR	0000028	0000028	0000028	0000028	0000028	0000028	0000028	SHR	0000028
SHRLI	0000029	0000029	0000029	0000029	0000029	0000029	0000029	SHRLI	0000029
ANDI	0000030	0000030	0000030	0000030	0000030	0000030	0000030	ANDI	0000030
ORI	0000031	0000031	0000031	0000031	0000031	0000031	0000031	ORI	0000031
XORI	0000032	0000032	0000032	0000032	0000032	0000032	0000032	XORI	0000032
SLLI	0000033	0000033	0000033	0000033	0000033	0000033	0000033	SLLI	0000033
SRLI	0000034	0000034	0000034	0000034	0000034	0000034	0000034	SRLI	0000034
SRLLI	0000035	0000035	0000035	0000035	0000035	0000035	0000035	SRLLI	0000035
SHR	0000036	0000036	0000036	0000036	0000036	0000036	0000036	SHR	0000036
SHRLI	0000037	0000037	0000037	0000037	0000037	0000037	0000037	SHRLI	0000037
ANDI	0000038	0000038	0000038	0000038	0000038	0000038	0000038	ANDI	0000038
ORI	0000039	0000039	0000039	0000039	0000039	0000039	0000039	ORI	0000039
XORI	0000040	0000040	0000040	0000040	0000040	0000040	0000040	XORI	0000040
SLLI	0000041	0000041	0000041	0000041	0000041	0000041	0000041	SLLI	0000041
SRLI	0000042	0000042	0000042	0000042	0000042	0000042	0000042	SRLI	0000042
SRLLI	0000043	0000043	0000043	0000043	0000043	0000043	0000043	SRLLI	0000043
SHR	0000044	0000044	0000044	0000044	0000044	0000044	0000044	SHR	0000044
SHRLI	0000045	0000045	0000045	0000045	0000045	0000045	0000045	SHRLI	0000045
ANDI	0000046	0000046	0000046	0000046	0000046	0000046	0000046	ANDI	0000046
ORI	0000047	0000047	0000047	0000047	0000047	0000047	0000047	ORI	0000047
XORI	0000048	0000048	0000048	0000048	0000048	0000048	0000048	XORI	0000048
SLLI	0000049	0000049	0000049	0000049	0000049	0000049	0000049	SLLI	0000049
SRLI	0000050	0000050	0000050	0000050	0000050	0000050	0000050	SRLI	0000050
SRLLI	0000051	0000051	0000051	0000051	0000051	0000051	0000051	SRLLI	0000051
SHR	0000052	0000052	0000052	0000052	0000052	0000052	0000052	SHR	0000052
SHRLI	0000053	0000053	0000053	0000053	0000053	0000053	0000053	SHRLI	0000053
ANDI	0000054	0000054	0000054	0000054	0000054	0000054	0000054	ANDI	0000054
ORI	0000055	0000055	0000055	0000055	0000055	0000055	0000055	ORI	0000055
XORI	0000056	0000056	0000056	0000056	0000056	0000056	0000056	XORI	0000056
SLLI	0000057	0000057	0000057	0000057	0000057	0000057	0000057	SLLI	0000057
SRLI	0000058	0000058	0000058	0000058	0000058	0000058	0000058	SRLI	0000058
SRLLI	0000059	0000059	0000059	0000059	0000059	0000059	0000059	SRLLI	0000059
SHR	0000060	0000060	0000060	0000060	0000060	0000060	0000060	SHR	0000060
SHRLI	0000061	0000061	0000061	0000061	0000061	0000061	0000061	SHRLI	0000061
ANDI	0000062	0000062	0000062	0000062	0000062	0000062	0000062	ANDI	0000062
ORI	0000063	0000063	0000063	0000063	0000063	0000063	0000063	ORI	0000063
XORI	0000064	0000064	0000064	0000064	0000064	0000064	0000064	XORI	0000064
SLLI	0000065	0000065	0000065	0000065	0000065	0000065	0000065	SLLI	0000065
SRLI	0000066	0000066	0000066	0000066	0000066	0000066	0000066	SRLI	0000066
SRLLI	0000067	0000067	0000067	0000067	0000067	0000067	0000067	SRLLI	0000067
SHR	0000068	0000068	0000068	0000068	0000068	0000068	0000068	SHR	0000068
SHRLI	0000069	0000069	0000069	0000069	0000069	0000069	0000069	SHRLI	0000069
ANDI	0000070	0000070	0000070	0000070	0000070	0000070	0000070	ANDI	0000070
ORI	0000071	0000071	0000071	0000071	0000071	0000071	0000071	ORI	0000071
XORI	0000072	0000072	0000072	0000072	0000072	0000072	0000072	XORI	0000072
SLLI	0000073	0000073	0000073	0000073	0000073	0000073	0000073	SLLI	0000073
SRLI	0000074	0000074	0000074	0000074	0000074	0000074	0000074	SRLI	0000074
SRLLI	0000075	0000075	0000075	0000075	0000075	0000075	0000075	SRLLI	0000075
SHR	0000076	0000076	0000076	0000076	0000076	0000076	0000076	SHR	0000076
SHRLI	0000077	0000077	0000077	0000077	0000077	0000077	0000077	SHRLI	0000077
ANDI	0000078	0000078	0000078	0000078	0000078	0000078	0000078	ANDI	0000078
ORI	0000079	0000079	0000079	0000079	0000079	0000079	0000079	ORI	0000079
XORI	0000080	0000080	0000080	0000080	0000080	0000080	0000080	XORI	0000080
SLLI	0000081	0000081	0000081	0000081	0000081	0000081	0000081	SLLI	0000081
SRLI	0000082	0000082	0000082	0000082	0000082	0000082	0000082	SRLI	0000082
SRLLI	0000083	0000083	0000083	0000083	0000083	0000083	0000083	SRLLI	0000083
SHR	0000084	0000084	0000084	0000084	0000084	0000084	0000084	SHR	0000084
SHRLI	0000085	0000085	0000085	0000085	0000085	0000085	0000085	SHRLI	0000085
ANDI	0000086	0000086	0000086	0000086	0000086	0000086	0000086	ANDI	0000086
ORI	0000087	0000087	0000087	0000087	0000087	0000087	0000087	ORI	0000087
XORI	0000088	0000088	0000088	0000088	0000088	0000088	0000088	XORI	0000088
SLLI	0000089	0000089	0000089	0000089	0000089	0000089	0000089	SLLI	0000089
SRLI	0000090	0000090	0000090	0000090	0000090	0000090	0000090	SRLI	0000090
SRLLI	0000091	0000091	0000091	0000091	0000091	0000091	0000091	SRLLI	0000091
SHR	0000092	0000092	0000092	0000092	0000092	0000092	0000092	SHR	0000092
SHRLI	0000093	0000093	0000093	0000093	0000093	0000093	0000093	SHRLI	0000093
ANDI	0000094	0000094	0000094	0000094	0000094	0000094	0000094	ANDI	0000094
ORI	0000095	0000095	0000095	0000095	0000095	0000095	0000095	ORI	0000095
XORI	0000096	0000096	0000096	0000096	0000096	0000096	0000096	XORI	0000096
SLLI	0000097	0000097	0000097	0000097	0000097	0000097	0000097	SLLI	0000097
SRLI	0000098	0000098	000009						

CS 110

Computer Architecture

Lecture 4: *Intro to Assembly Language, RISC-V Intro*

Video 2: Memory Operations

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

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`

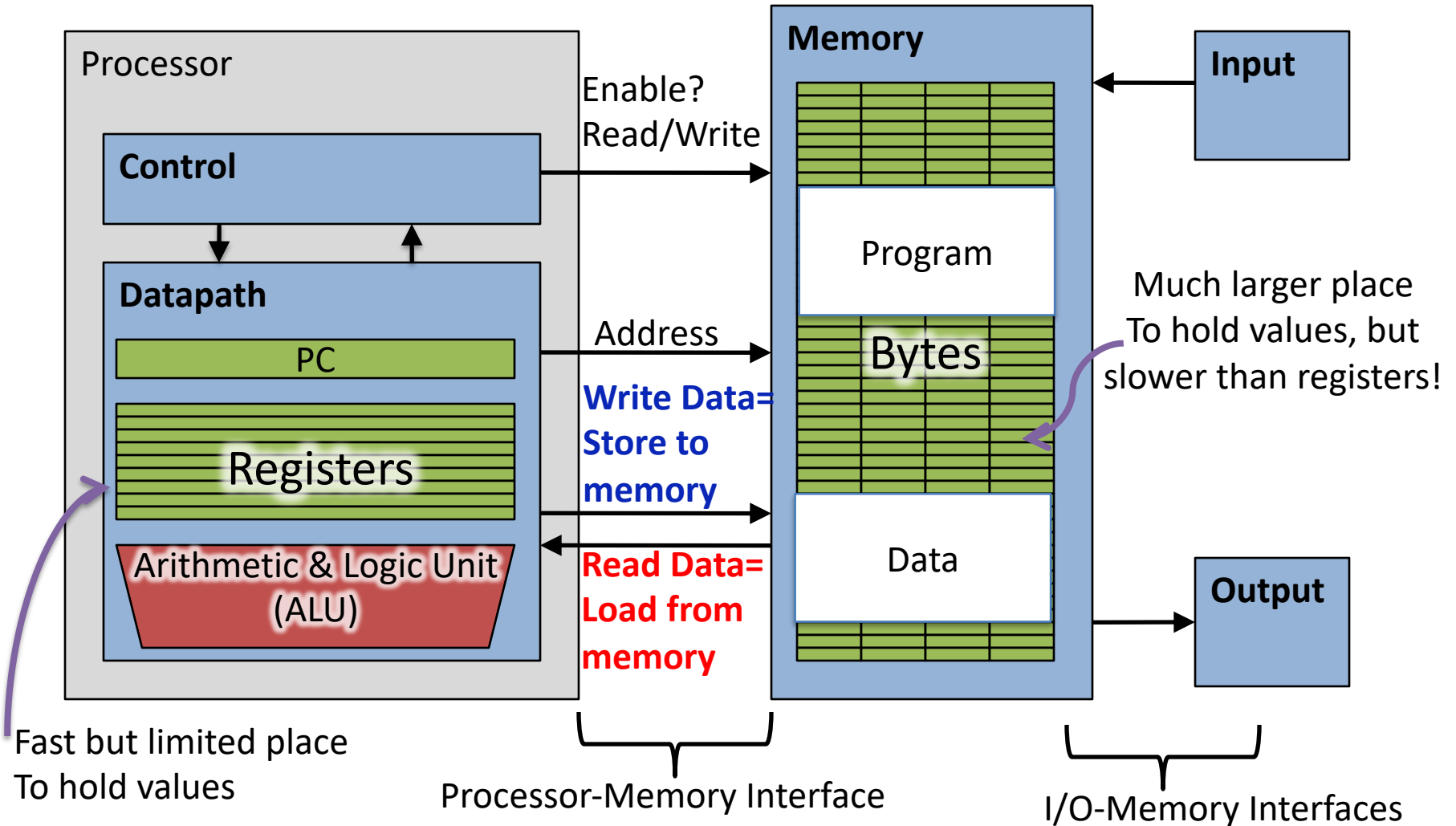
Register Zero

- 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!

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

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 lowest byte

Little Endian: Start with the small end (little end; Least significant byte of the word) ↓

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

bit: 31 24 23 16 15 8 7 0

Big Endian: Start with the big end (Most significant byte) ↑

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., 2020-03-22)

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., 22/03/2020)

Eating Pizza skinny part first (the normal way)

CANopen

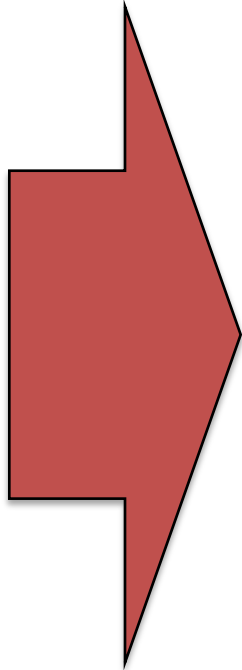
Intel x86; RISC-V

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

Example

Memory

Addr. dec	Addr. hex	8-bit Value
...
15	0x0F	0x77
14	0x0E	0x66
13	0x0D	0x55
12	0x0C	0x44
11	0x0B	0x33
10	0x0A	0x22
9	0x09	0x11
8	0x08	0x00
7	0x07	0xEF
6	0x06	0xCD
5	0x05	0xAB
4	0x04	0x89
3	0x03	0x67
2	0x02	0x45
1	0x01	0x23
0	0x00	0x01



Addresses (hex):

...
F	E	D	<u>C</u>
B	A	9	<u>8</u>
7	6	5	<u>4</u>
3	2	1	<u>0</u>

Little Endian

Word at address 0x0C: 0x 77 66 55 44

Word at address 0x08: 0x 33 22 11 00

Word at address 0x04: 0x EF CD AB 89

Word at address 0x00: 0x 67 45 23 01

...
77	66	55	44
33	22	11	00
EF	CD	AB	89
67	45	23	01

Addresses (hex):

...
<u>C</u>	D	E	F
<u>8</u>	9	A	B
<u>4</u>	5	6	7
<u>0</u>	1	2	3

Big Endian

Word at address 0x0C: 0x 44 55 66 77

Word at address 0x08: 0x 00 11 22 33

Word at address 0x04: 0x 89 AB CD EF

Word at address 0x00: 0x 01 23 45 67

...
44	55	66	77
00	11	22	33
89	AB	CD	EF
01	23	45	67

RISC-V: Little Endian

(E.g., $1025 = 0x401 = 0b\ 0100\ 0000\ 0001$)

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

- Hexadecimal number:
 $0xFD34AB88$ ($4,248,087,432_{ten}$) \Rightarrow
 - 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:	67	66	54	64
Data:	0xFD	0x34	0xAB	0x88

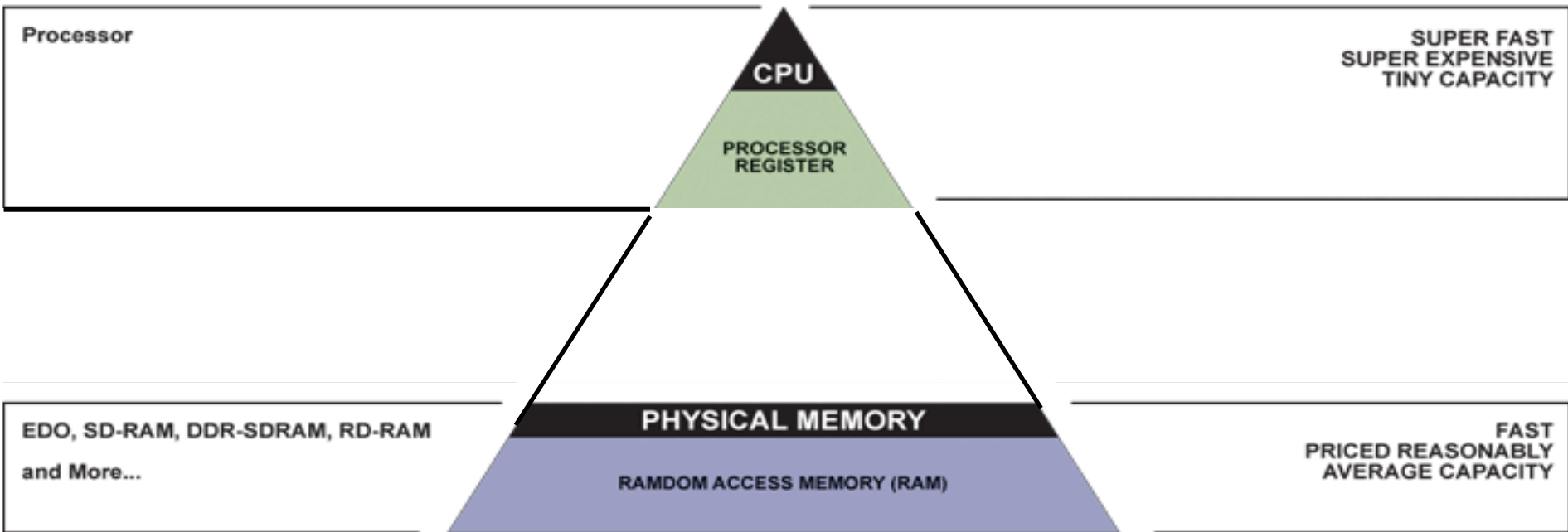
- Little Endian: Starts with the little end of a word:
 - It starts with the smallest (least significant) Byte

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

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

bit: 31 24 23 16 15 8 7 0

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];          /* A => x15 */
g = h + A[3];        /* h => x13 */
```



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

```
lw  x10, 12 (x15) # Reg x10 gets A[3]
add x11, x13, 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 => x15 */
A[10] = h + A[4]; /* h => x13 */
```

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

```
lw   x10, 12(x15)  # Temp reg x10 gets A[3]
add  x10, x13, x10  # Temp reg x10 gets h + A[3]
sw   x10, 40(x15)  # A[10] = h + A[3]
```



Note: x15 – base register (pointer)
 12, 40 – offsets in bytes

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

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**?



Question! What's in **x12**?

```
addi x11, x0, 0x4F6
```

```
sw x11, 0(x5)
```

```
lb x12, 1(x5)
```



A:	0x0
B:	0x4
C:	0x6
D:	0xF
E:	0xFFFFFFFF

Question! What's in **x12**?

```
addi x11, x0, 0x85BCF6
```

```
sw x11, 0(x5)
```

```
lb x12, 2(x5)
```



A:	0x8
B:	0x85
C:	0xC
D:	0xBC
E:	0xFFFFFFFF8
F:	0xFFFFFFFF85
G:	0xFFFFFFFFC
H:	0xFFFFFFFFBC

Question!

Which of the following is TRUE?

- A: `add x10, x11, 4(x12)` is valid in RV32
- B: can byte address 8GB of memory with an RV32 word
- C: `imm` must be multiple of 4 for `lw x10, imm(x10)` to be valid

– D: None of the above



“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
- RISC-V is Little Endian
- Instructions:
add, addi, sub, lw, sw, lb, lbu, sw
- Registers:
 - 32 registers, referred to as x0 – x31
 - Zero: x0

Question:

Piazza: "Lecture 4 RISC-V poll"

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)`