EE 044252: Digital Systems and Computer Structure

Winter 2018

Lecture 1: *Course Introduction and RISC-V Architecture*

Instructors:

Shahar Kvatinsky, Nir Katzir, Reuven Dobkin

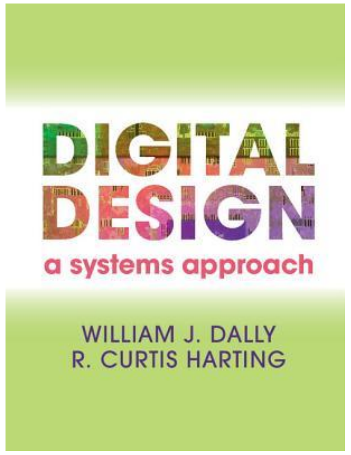
`Find us on Moodle`

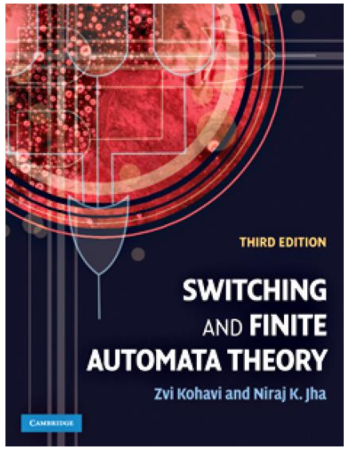
Following…

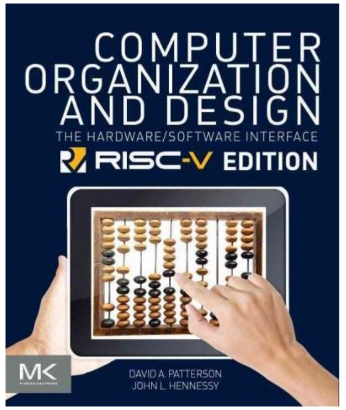
- Berkeley CS61C by Randy Katz and Krste Asanović
- Stanford EE108 by Subhasish Mitra and Bill Dally
- Technion 044145/234145 and 044262/234262
- And the rest of the world

Course Information

- Course Web on moodle.technion.ac.il
- Instructors: Shahar Kvatinsky, Nir Katzir, Reuven Dobkin
- Teaching Staff:
  - Head TA: Maroun Tork
  - TAs: Ben Perach, Ori Linial, Rotem Ben Hur
  - Verilog Workshops: Adi Eliahu, Leonid Azriel
  - Simulations: Moshe Kimhi, Basel Diab
- Textbooks:
  - Dally, *Digital Design*
  - Kohavi and Jha, *Switching and Finite Automata Theory*
  - Patterson & Hennessey, *Computer Organization and Design*, **RISC-V ed.**




THIRD EDITION
SWITCHING AND FINITE AUTOMATA THEORY
Zvi Kohavi and Niraj K. Jha


COMPUTER ORGANIZATION AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
RISC-V EDITION
MK
DAVID A. PATTERSON
JOHN L. HENNESSY

More Course Information

- Midterm (M): 15.05.2019 (magen), 20%
- Final exam (E) – A-term: 04.07.2019
- Final exam (E) – B-term: 07.10.2019
- Quizzes (Q) – 3% (compulsory)
- Solve homework first, come to tutorials second
  - We solve only some of the questions in tutorials
  - Some tutorial sessions teach new stuff
- Workshops on Verilog (Hardware Description Language) details
- Simulations (S) in pairs (27%)
- Final grade: E > 54 ? max(E+S+Q+M, E+S+Q) : E

EE 044252: Digital Systems and Computer Structure

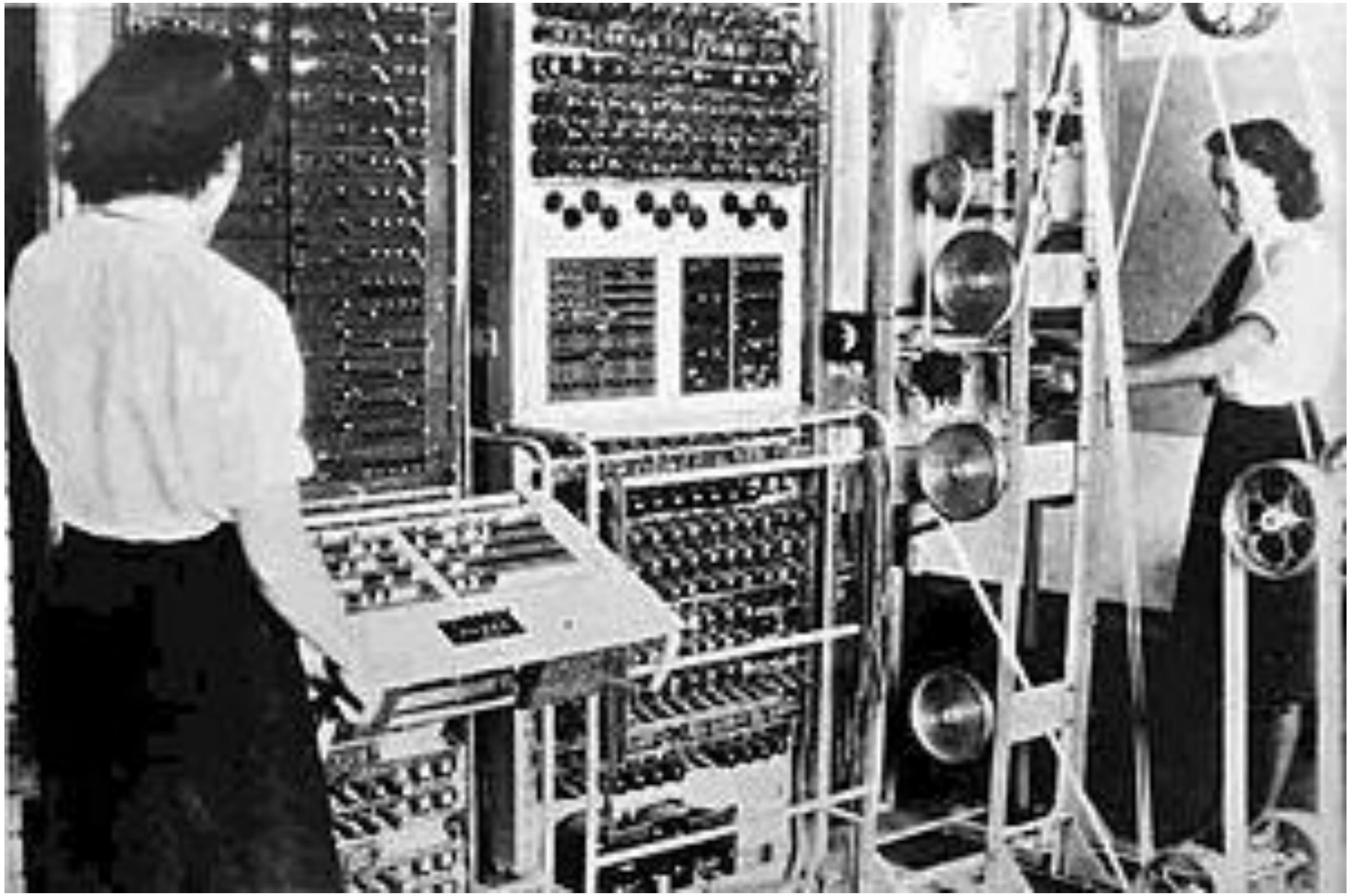
Topic	wk	Lectures	Tutorials	Workshop	Simulation
Arch	1	Intro. RISC-V architecture	Numbers. Codes		
Comb	2	Switching algebra & functions	Assembly programming		
	3	Combinational logic	Logic minimization	Combinational	
	4	Arithmetic. Memory	Gates		Combinational
Seq	5	Finite state machines	Logic		
	6	Sync FSM	Flip flops, FSM timing	Sequential	Sequential
	7	FSM equiv, scan, pipeline	FSM synthesis		
	8	Serial comm, memory instructions	Serial comm, pipeline		
µArch	9	Function call, single cycle RISC-V	Function call		
	10	Multi-cycle RISC-V	Single cycle RISC-V		Multi-cycle
	11	Interrupts, pipeline RISC-V	Multi-cycle RISC-V		
	12	Dependencies in pipeline RISC-V	Microcode, interrupts		
	13		Depend. in pipeline RISC-V		

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion



1944: Colossus: world's first electronic **programmable** digital computer; used by British codebreakers during World War II

(recall Enigma)

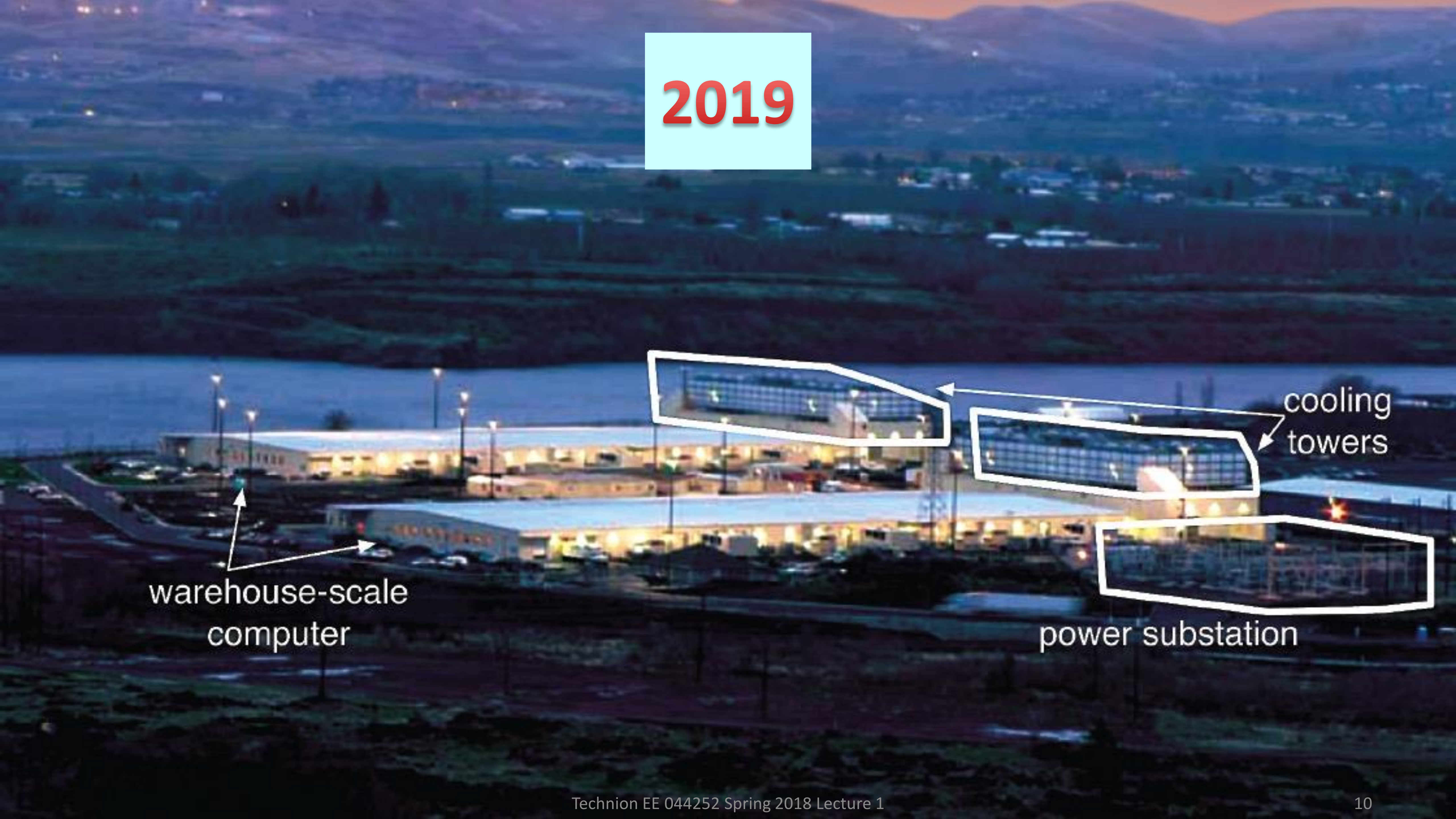
*programmed by switches and plugs and not by a stored program

2019

Personal Mobile Devices

Network Edge Devices

2019

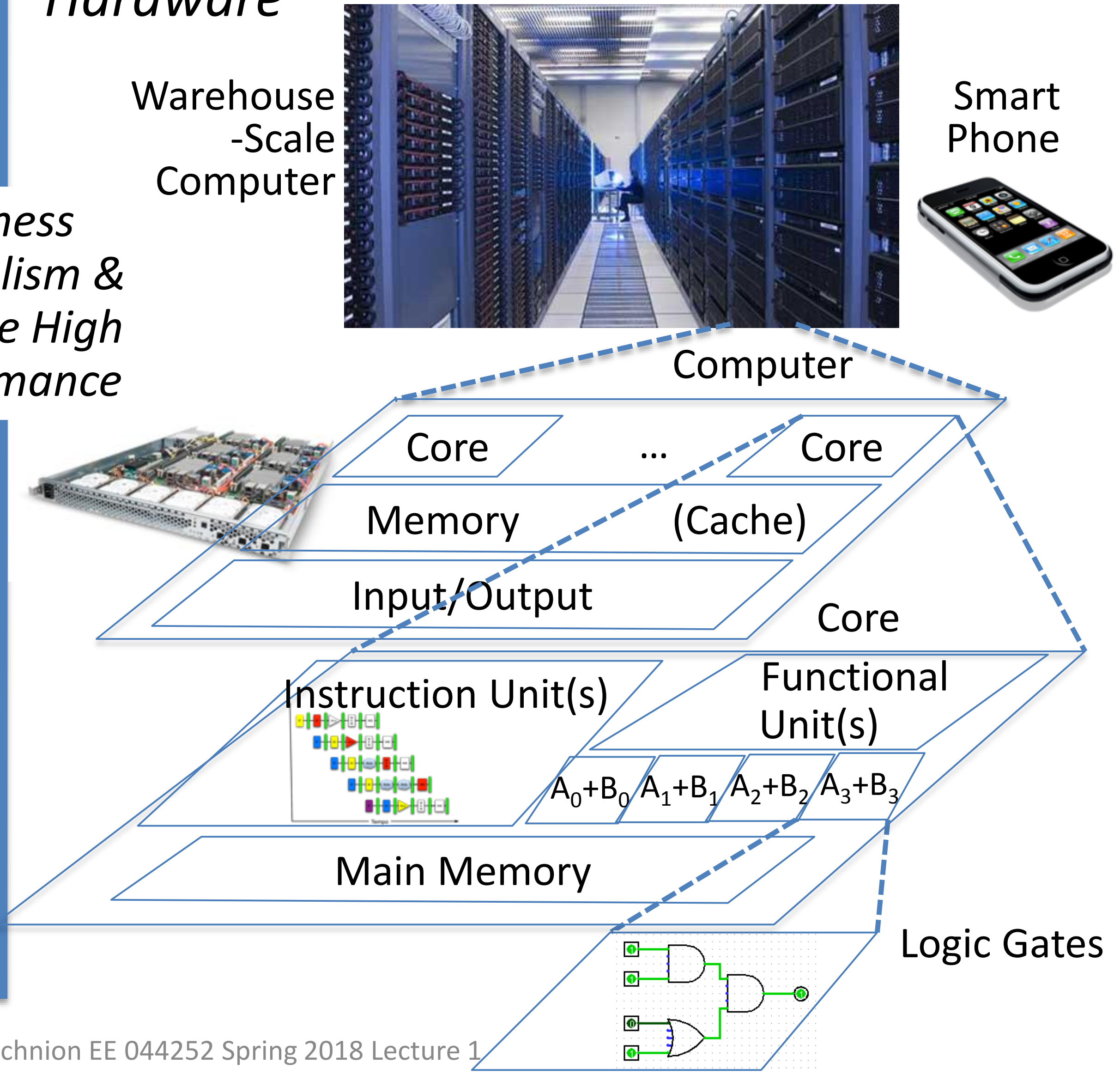

cooling towers
warehouse-scale computer
power substation

2018 Machine Structures

Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "cats"
- Parallel Threads
  Assigned to core (CPU)
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

Harness Parallelism & Achieve High Performance



Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Seven Great Ideas in Computer Architecture

Fundamental:

1. The discrete (“digital”) approach to representing information
2. The stored-program (“Von Neumann”) architecture

Practicality:

3. Abstraction (Layers of Representation/Interpretation)

Performance and dependability:

4. Moore’s Law (Exploit technology trends)
5. Principle of Locality (Memory Hierarchy)
6. Parallelism
7. Dependability via Redundancy

#1: Discrete Representation (“Digital”) + Regeneration

- Continuous-value representation is very efficient:
  - the exact value of a voltage, translated to a number, can represent an “infinite” amount of information, **BUT**
- Cannot be passed on reliably
- Discrete (digital):
  - There is only a small (2 in binary representation) number of “legal” values
  - A received value is “corrected” to the nearest legal value (regeneration)
  - Less efficient (theoretically), but works even in lousy components!

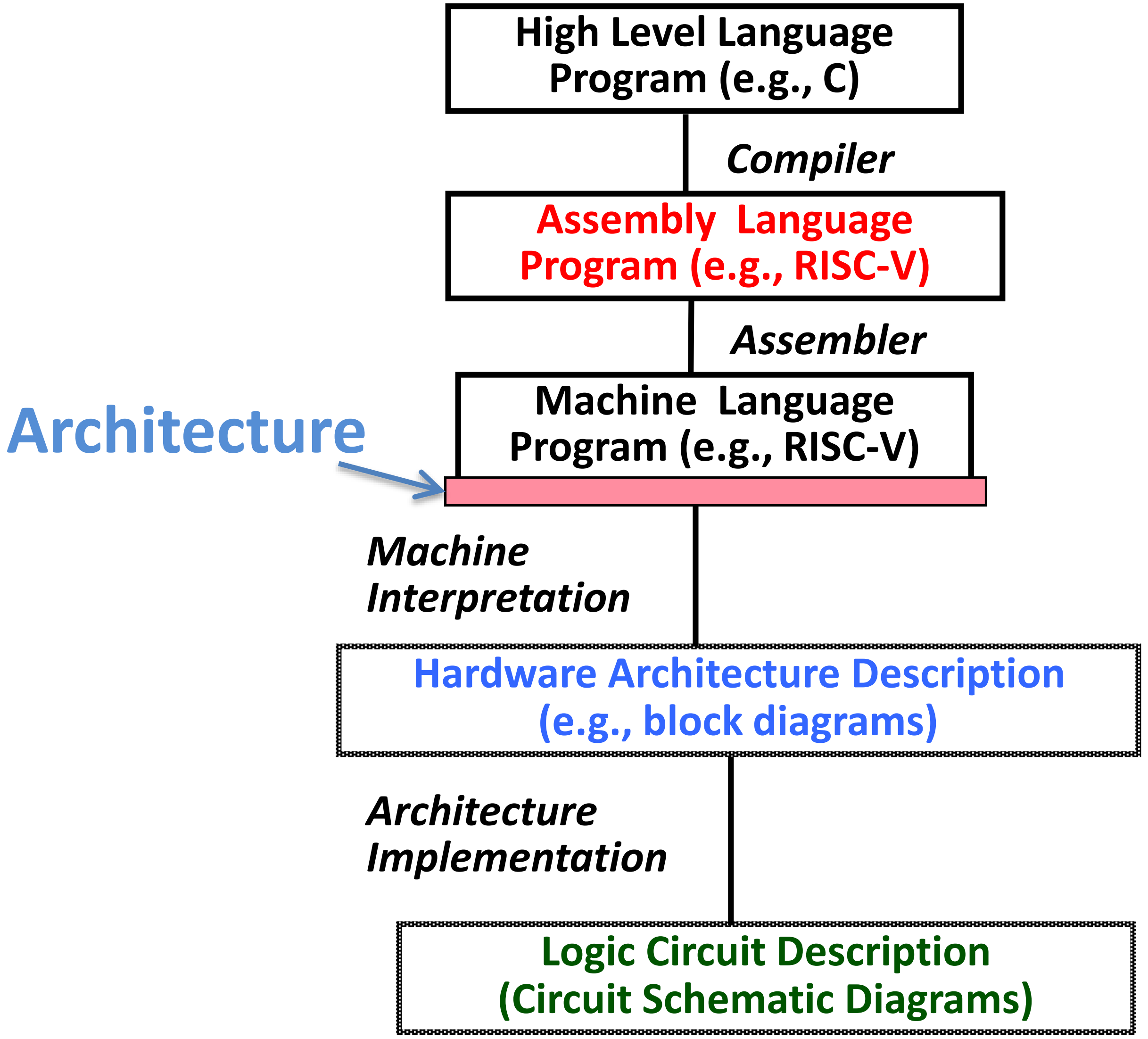
#2: The Stored-Program ("Von Neumann") Architecture

- The hardware ("processor") is fixed
  - efficient for manufacturing + low cost made possible by high volume
- It repeatedly reads instructions from memory and executes them
  - can do anything, just need to change the program

- Remark: there are alternative approaches
  - Hardware that can be reconfigured for a desired computation
    - flexibility by reconfiguration of the computing hardware
    - Example: Field Programmable Gate Arrays (FPGA)
  - In-memory (including associative) computing

Important in 044252

Great Idea #3: Abstraction (Levels of Representation/Interpretation)

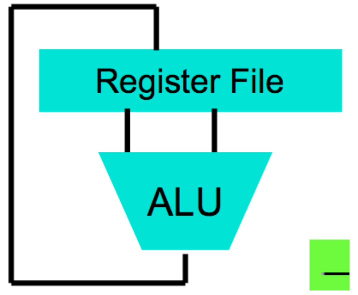

High Level Language Program (e.g., C)
Compiler
Assembly Language Program (e.g., RISC-V)
Assembler
Machine Language Program (e.g., RISC-V)
Architecture
Machine Interpretation
Hardware Architecture Description (e.g., block diagrams)
Architecture Implementation
Logic Circuit Description (Circuit Schematic Diagrams)

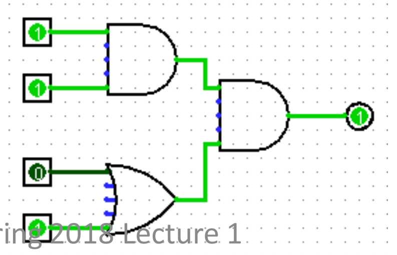
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

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
```





Secondary in 044252

#4: Moore's Law

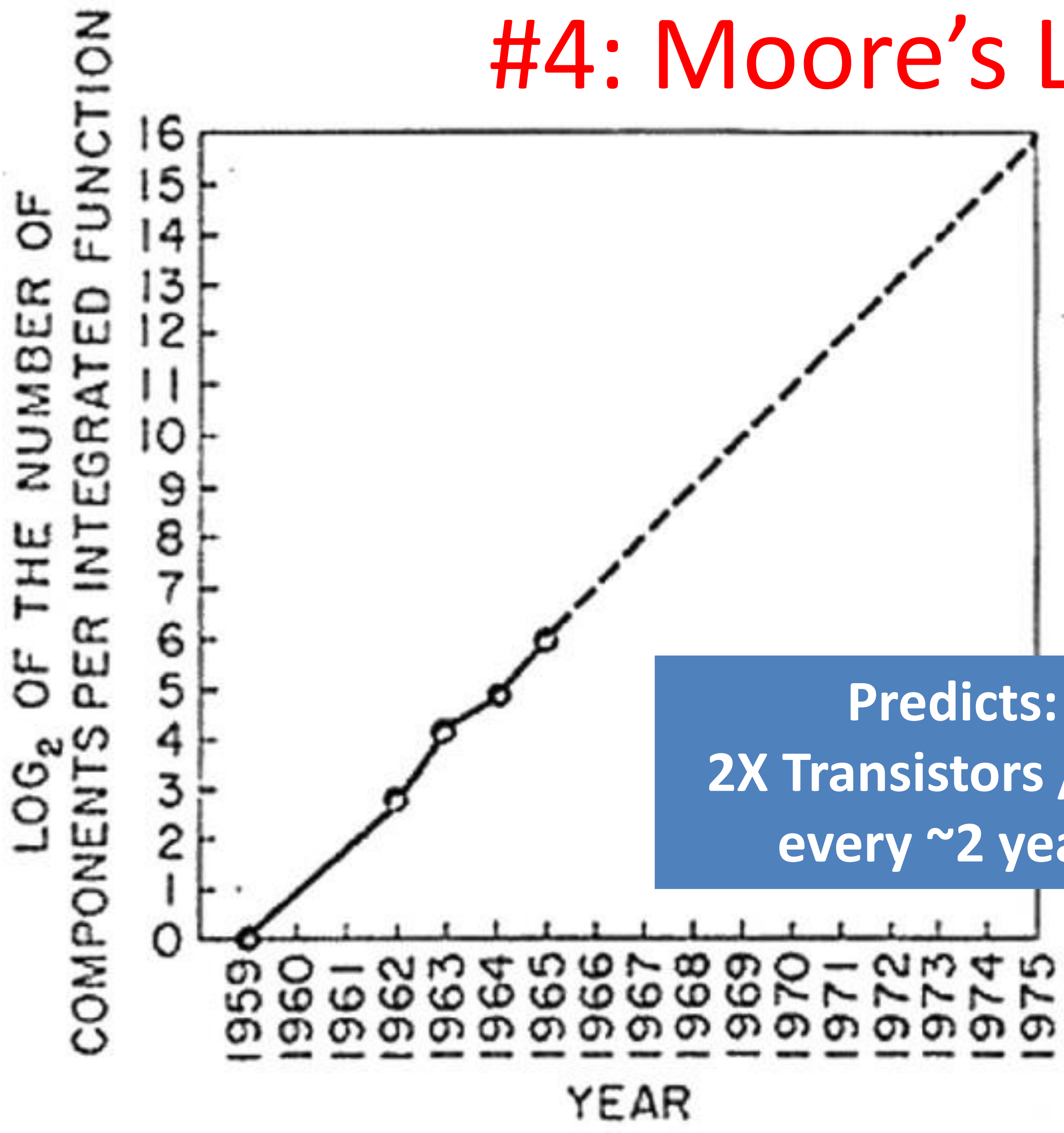
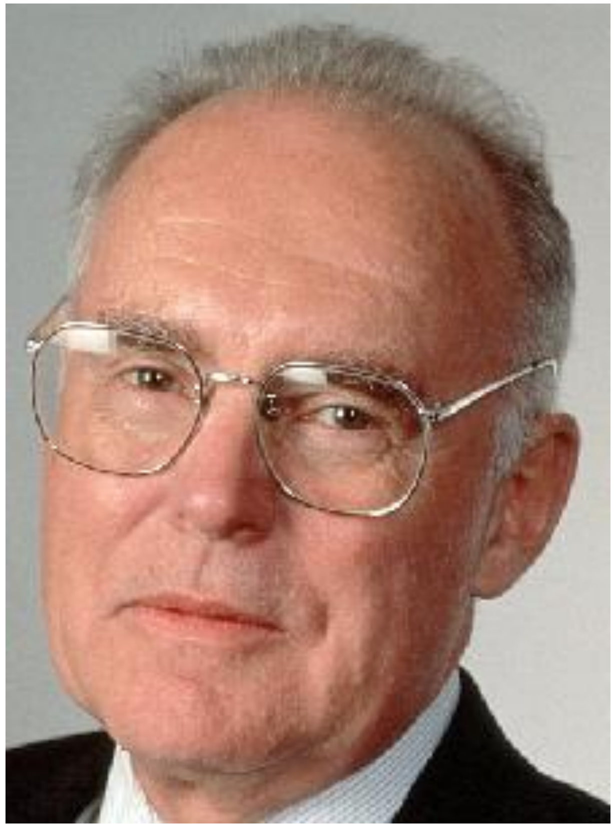


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

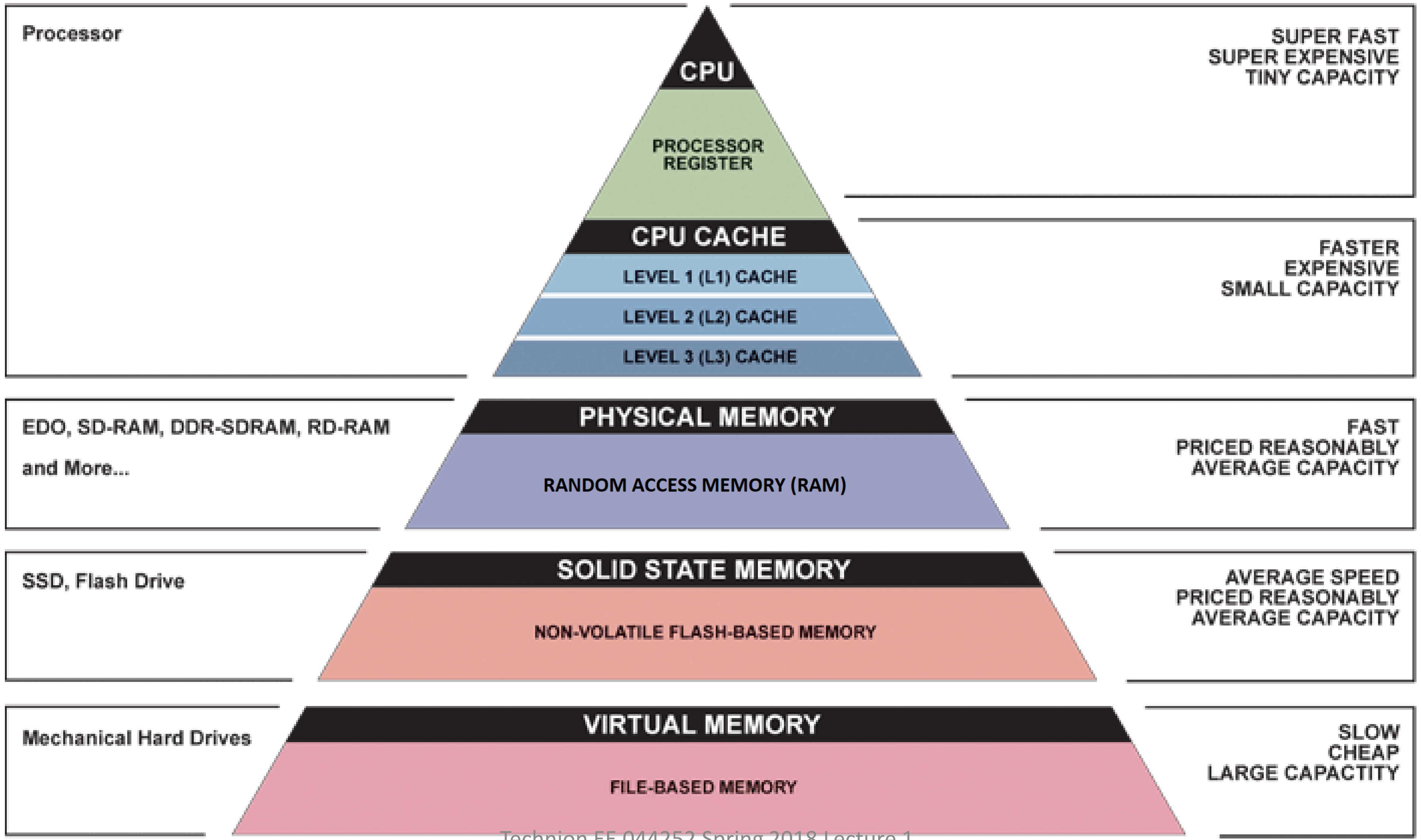
Predicts:
2X Transistors / chip every ~2 years



Gordon Moore
Intel Cofounder

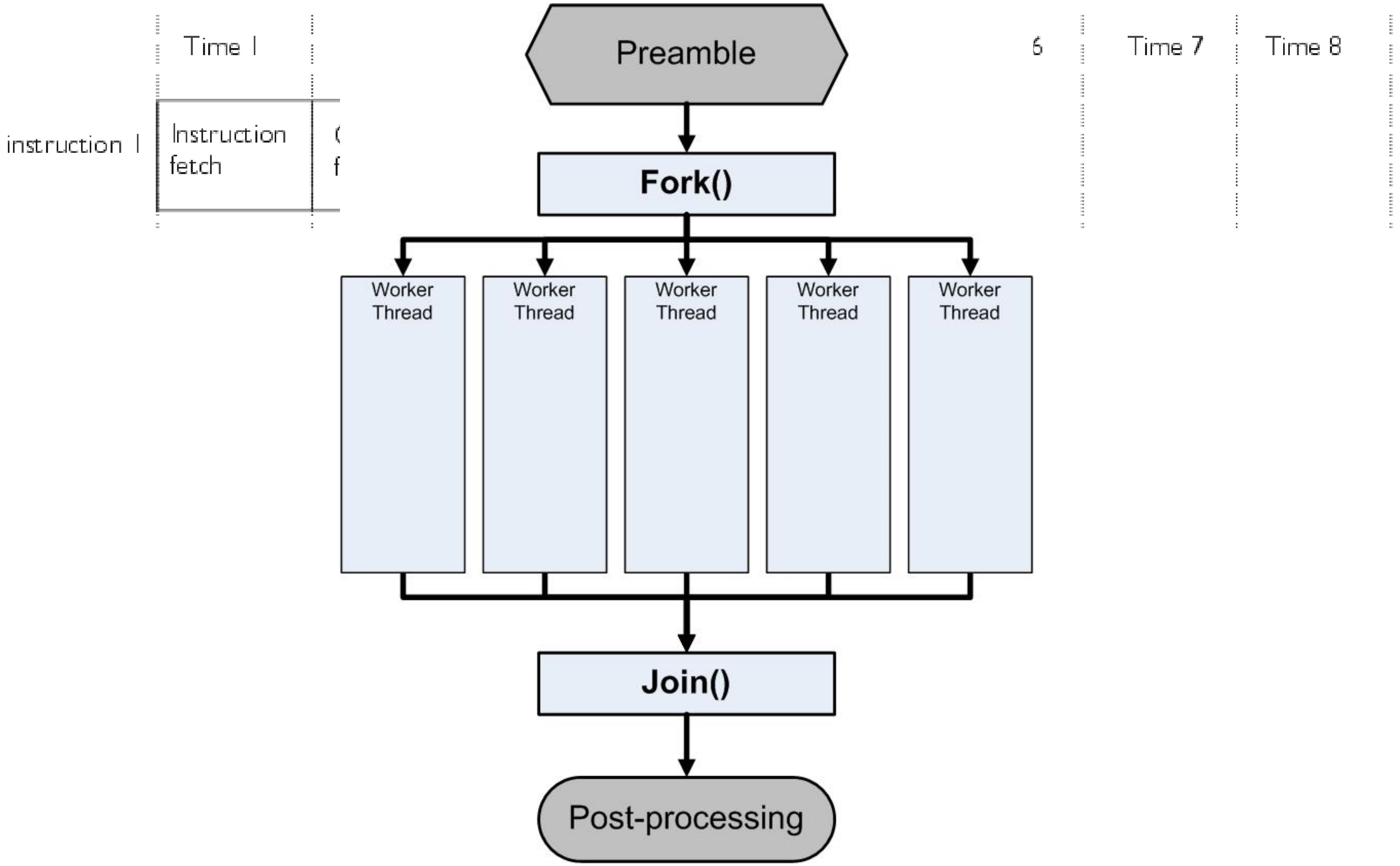
Secondary in 044252

Great Idea #5: Principle of Locality/ Memory Hierarchy



Important in 044252

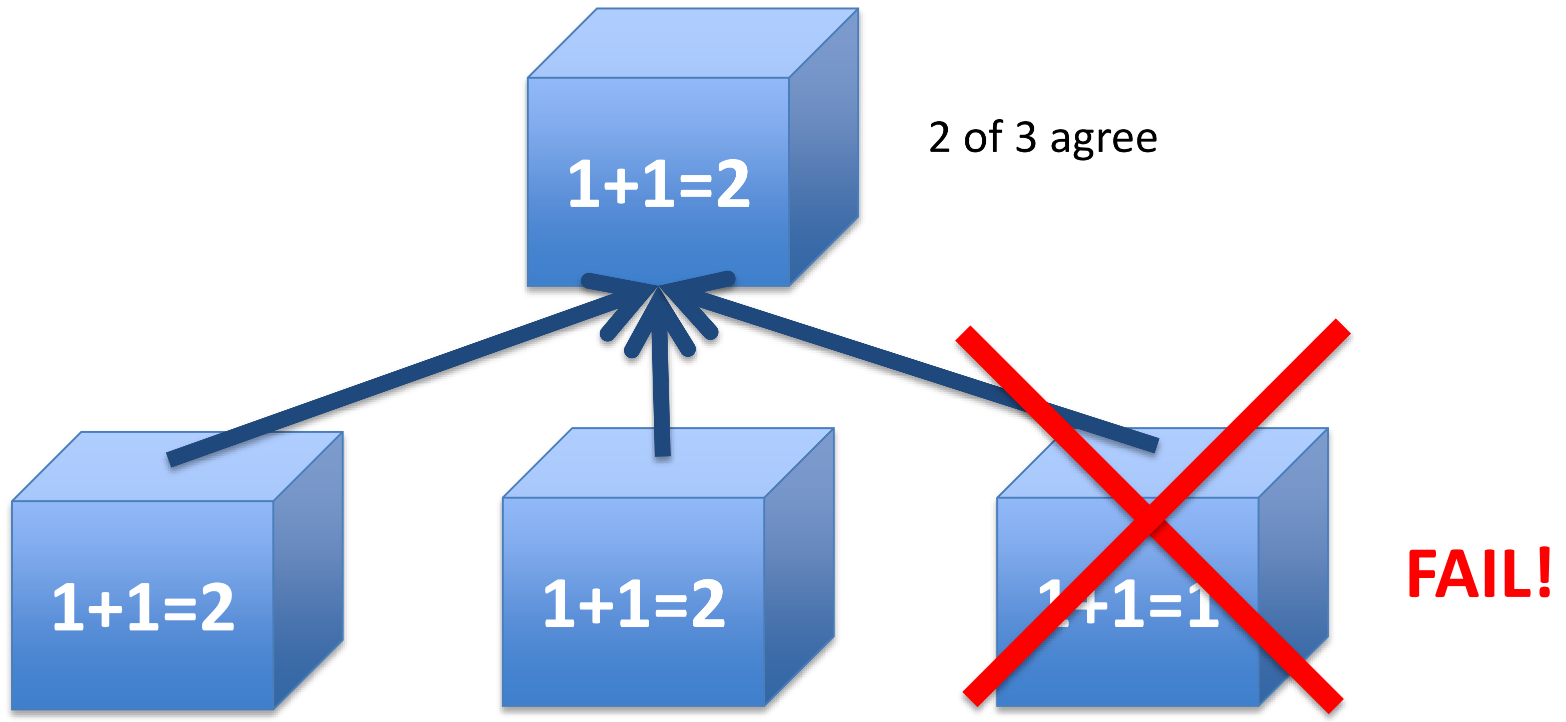
Great Idea #6: Parallelism



Secondary in 044252

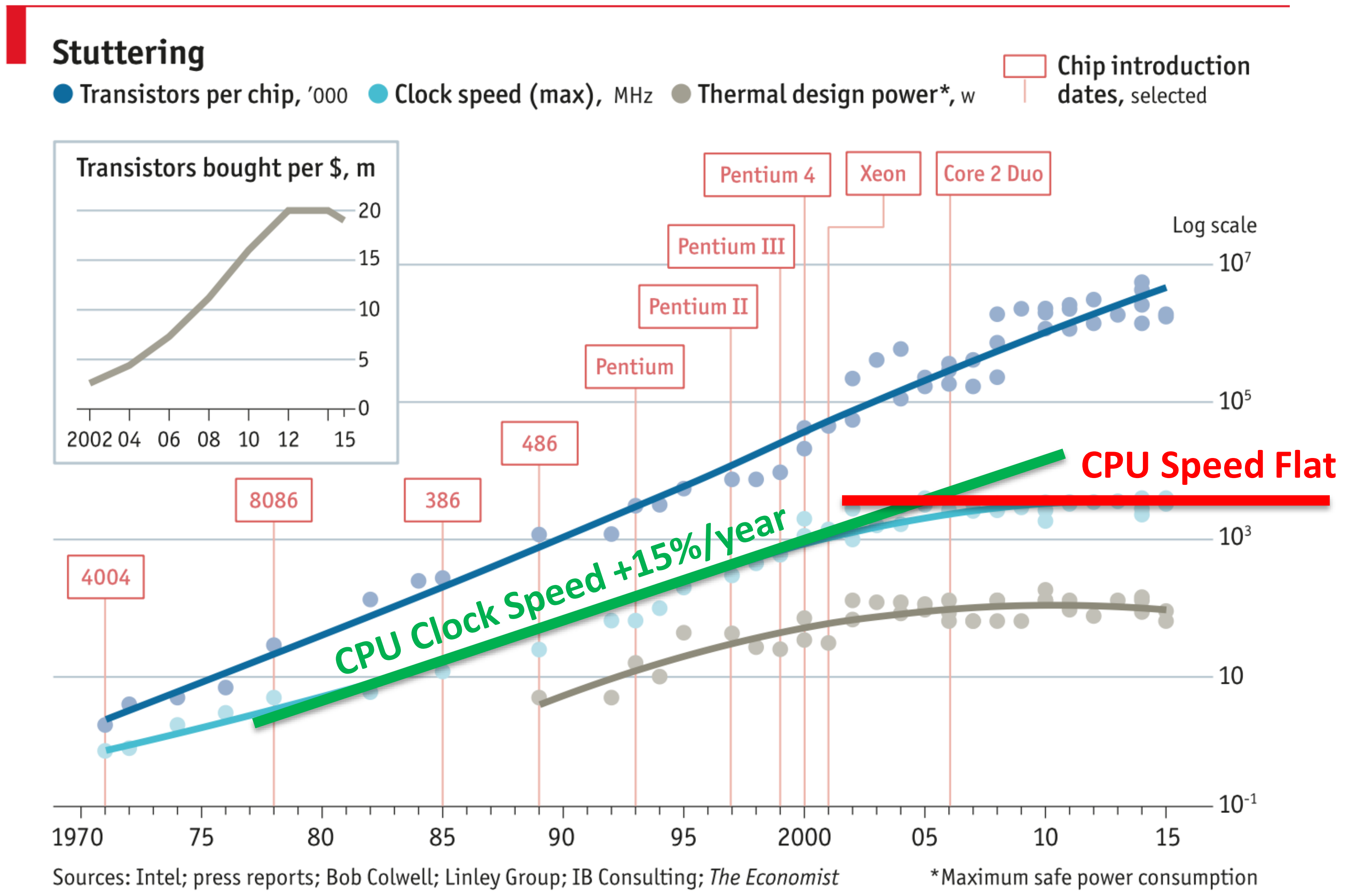
Great Idea #7: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Why are Logic and Architecture Exciting Today?

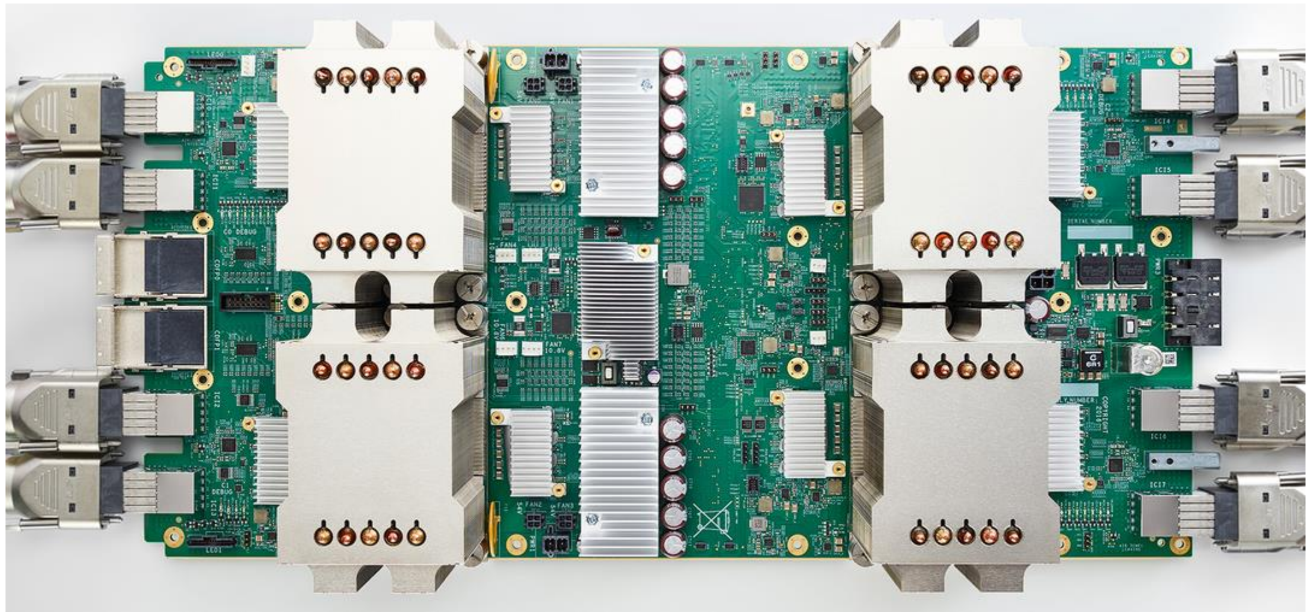


Old Conventional Wisdom

- Moore’s Law = faster, cheaper, lower-power general-purpose computers each year
- In glory days (pre-2000), 1%/week performance improvement!

- Dumb to compete by designing parallel or specialized computers
- By time you’ve finished design, next generation of general-purpose will beat you

New Conventional Wisdom



Google TPU2
(Tensor Processing Unit)
Specialized Engine for NN training
Deployed in cloud
45 TFLOPS/chip

Serious heatsinks!

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Levels of Representation/Interpretation

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

Compiler

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

Assembler

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

Machine Interpretation

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

Architecture Implementation

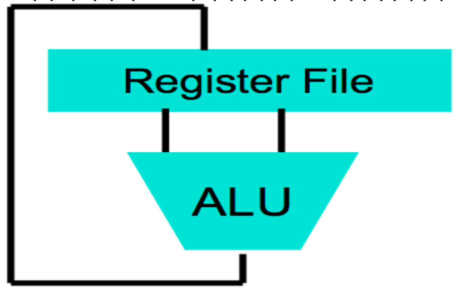
Logic Circuit Description (Circuit Schematic Diagrams)

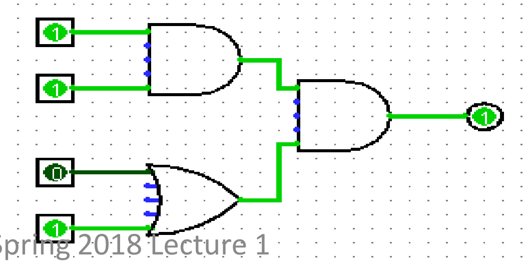
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

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)

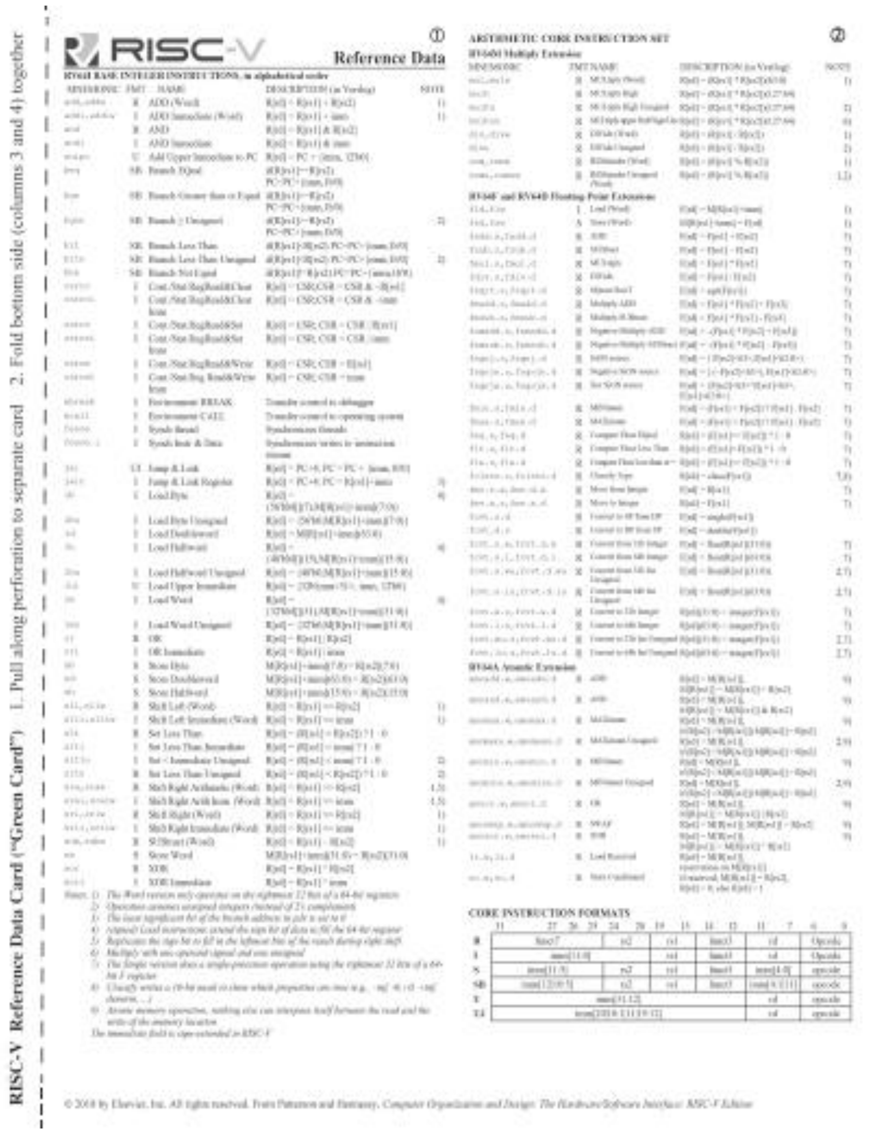
- Job of a CPU (*Central Processing Unit*, aka *Core*): execute *instructions*
- Instructions: CPU's primitives operations
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence …
  - With additional operations to change the sequence
- CPUs belong to "families," each 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
  - VAX architecture had an instruction to multiply polynomials!
- 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 software do complicated operations by composing simpler ones

RISC-V “Green Card”

On the Moodle site



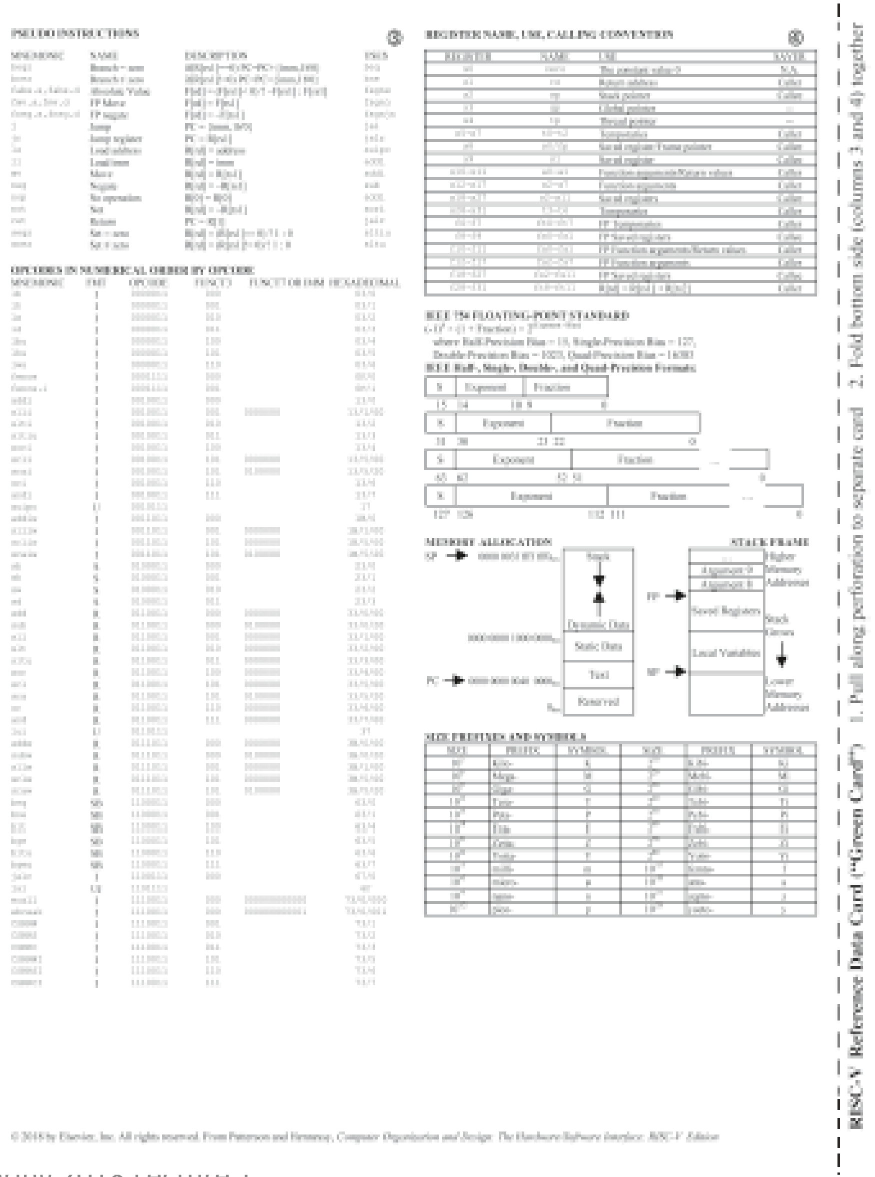
RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

ARITHMETIC CORE INSTRUCTION SET

CORE INSTRUCTION FORMATS

[illegible]



PSEUDO INSTRUCTIONS

OPCODES IN NUMERICAL ORDER BY OPCODE

REGISTER NAME, USE, CALLING CONVENTION

IEEE 754 FLOATING-POINT STANDARD

MEMORY ALLOCATION

STACK FRAME

SIZE PREFIXES AND SYMBOLS

[illegible]

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

RISC-V

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
- Both proprietary and open-source core implementations
- 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 use 32-bit (RV32) in class, textbook uses 64-bit (RV64)
- Standard maintained by non-profit RISC-V Foundation

Foundation Members (60+)

Platinum:

Berkeley Architecture Research

bluespec

cortus

DRAPER

IBM

Google

SAMSUNG

Microsemi

Mellanox TECHNOLOGIES

QUALCOMM

Microsoft

SiFive

NVIDIA

Rambus Cryptography Research

Western Digital

HUAWEI

Gold, Silver, Auditors:

AMD

ANDES TECHNOLOGY

antmicro EMBEDDED SYSTEMS

Blockstream

ICT 中科院计算所 INSTITUTE OF COMPUTING TECHNOLOGY, CAS

codasip

Esperanto Technologies

ESPRESSIF

GRAY RESEARCH

MEDIATEK

IDT

INTRINSIX

ETH zürich

LATTICE SEMICONDUCTOR

lowRISC

PROCESSOR

MIT CSAIL

ROA LOGIC

Rumble Development

runtime.io

Sur Technology

Syntacore Custom cores and tools

Technolution

VectorBlox embedded supercomputing

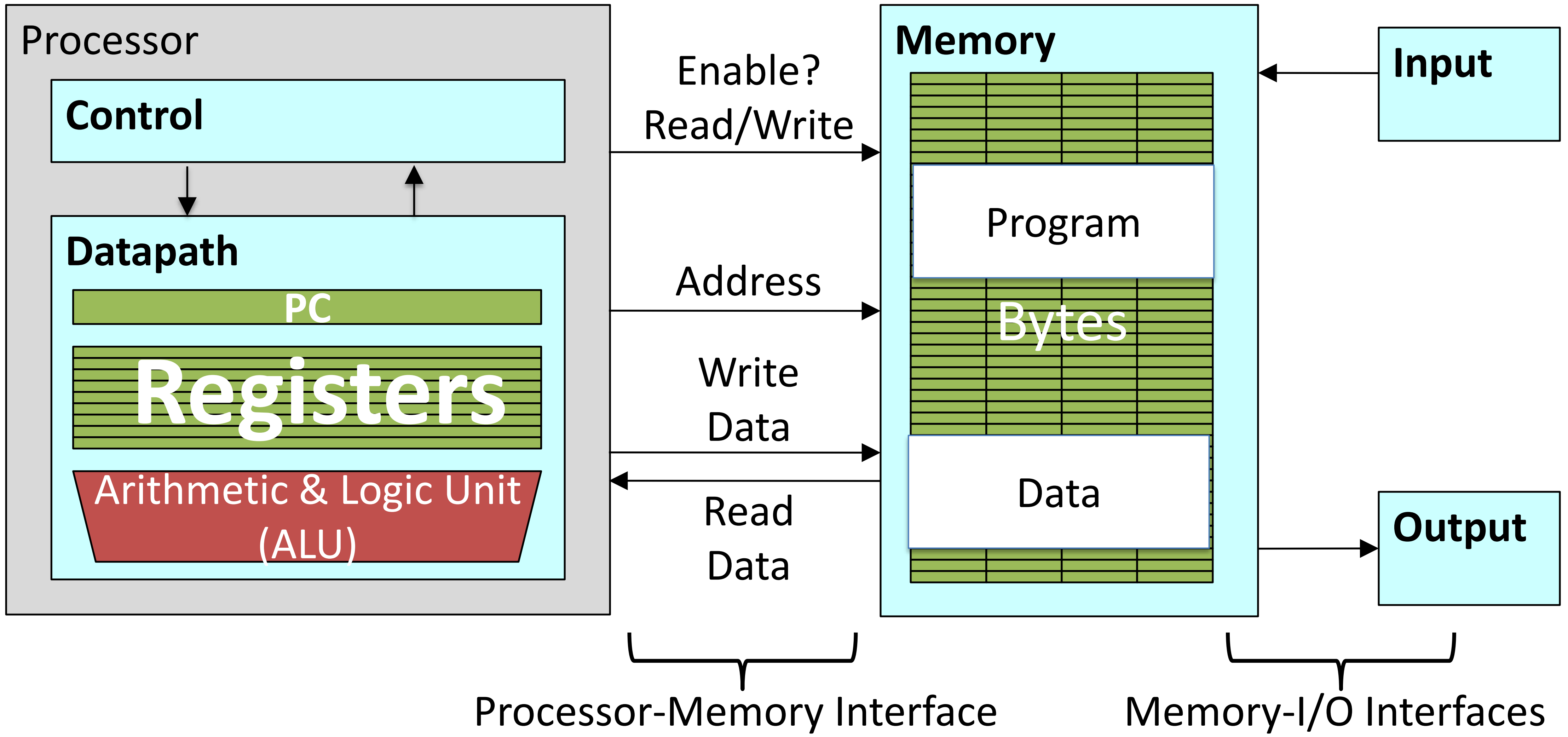
Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have *variables* as you know and love them
  - More primitive, closer to what simple hardware can directly support
- Assembly 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 (faster than 1 ns - light travels 1 foot in 1 ns!!!)

Registers live inside the Processor



How many RISC-V Registers ?

- Drawback: Since registers are in hardware, there are a limited number of them
  - Solution: program must be carefully written to efficiently use registers
- 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
  - Groups of 32 bits called a word in RISC-V ISA
    - P&H CoD textbook uses the 64-bit variant RV64 (explain differences later)
    - In class we use RV32 and registers are 32 bits each
- **`x0`** is special, always holds value zero
  - So really only 31 registers able to hold variable values

C 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

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

RISC-V Instruction Assembly Syntax

- Instructions have an opcode and operands
- E.g., **`add x1, x2, x3`**   **`# x1 = <x2> + <x3>`**

# is assembly comment syntax

Second operand register

Destination register

Operation code (opcode)

First operand register

<x2> means contents of x2

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

Example

- 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**

Immediates

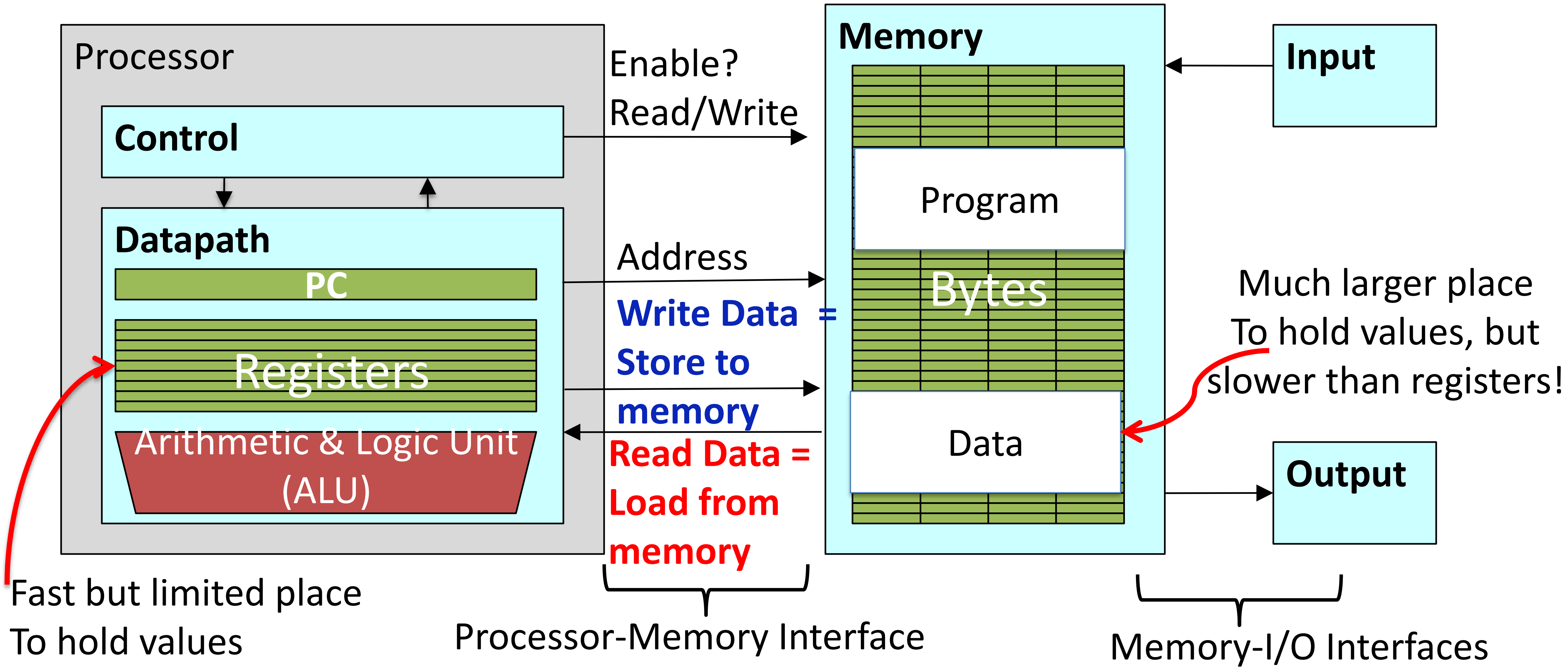
- Immediates are numerical constants
- They appear often in code, so there are special instructions for them
- Add Immediate:

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

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

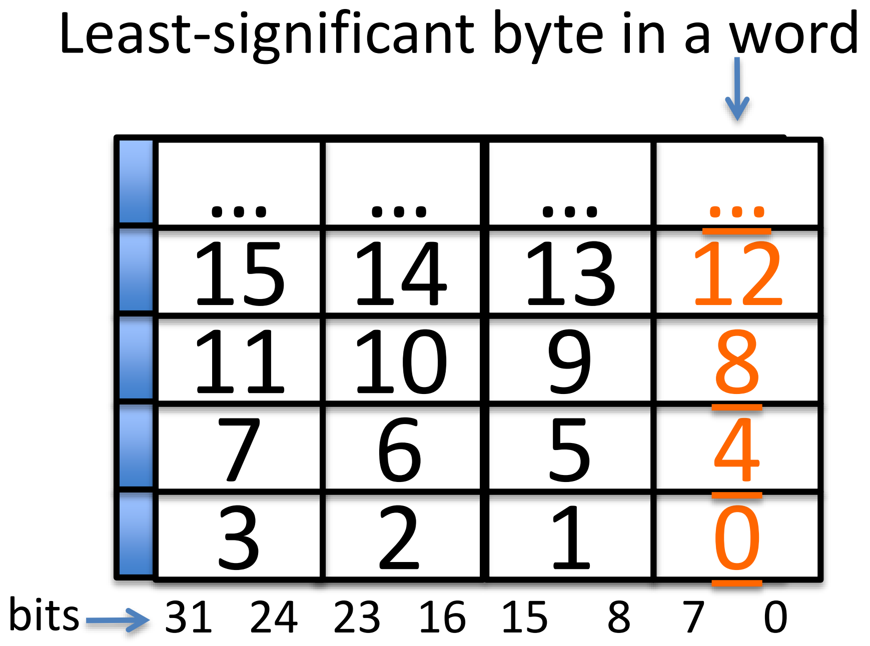
```
f = g                   (in C)
add x3,x4,x0            (in RISC-V)
```

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
Program
Bytes
Data
Input
Output
Much larger place To hold values, but slower than registers!
Fast but limited place To hold values
Processor-Memory Interface
Memory-I/O Interfaces

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
- 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)

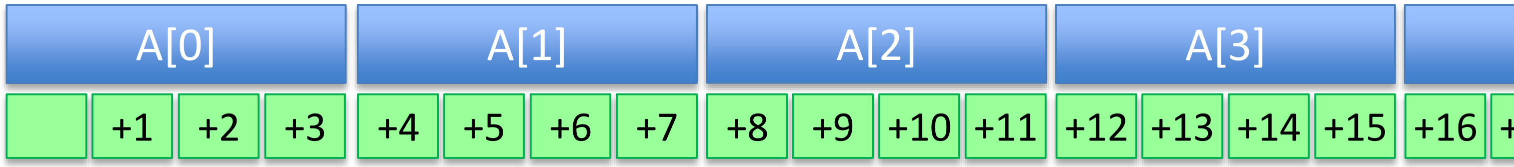


Least-significant byte gets the smallest address

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]
```

x13–base register, pointer to A[0])
12 is offset in bytes
12 = 3 (index) × 4 (sizeof(int))

```
add x11,x12,x10  # g = h + A[3]
```

Transfer from Register to Memory

- C code

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

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

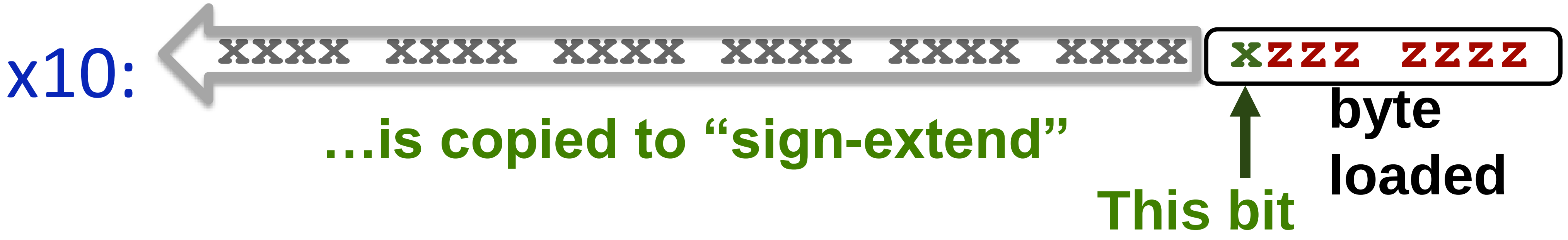
```
lw   x10,12(x13)     # Temp reg x10 gets A[3]
add  x10,x12,x10     # Temp reg x10 gets h + A[3]
sw   x10,40(x13)     # A[10] = h + A[3]
```

Note: `x13` – base register (pointer)
`12,40` – offsets in bytes
x13+12 and x13+40 must be multiples of 4 (why?)

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" load (**`lbu`**) which zero extends to fill register. Why no unsigned store byte **`sbu`**?



Your turn

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

Answer	x12
RED	0x5
GREEN	0xf
ORANGE	0x3
YELLOW	0xffffffff

The HexaDecimal Code

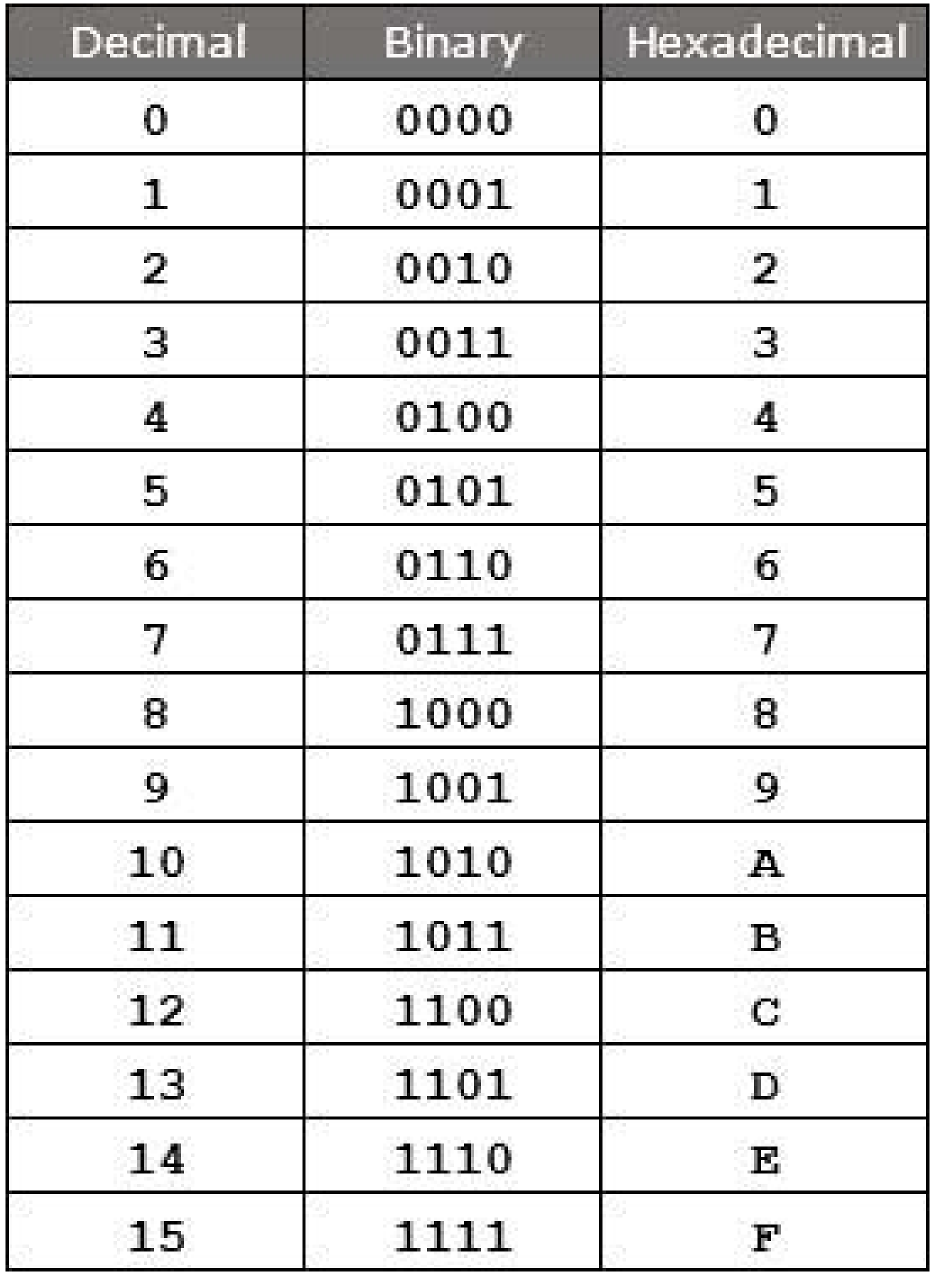
0x 3 f 5

0b 0011 1111 0101

32-bit:

0x 00 00 03 f5

0b 0000 0000 0000 0011 1111 0101



Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Your turn

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

Answer	x12
RED	0x5
GREEN	0xf
[✓ ORANGE]	[✓ 0x3]
YELLOW	0xffffffff

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Instructions we learnt so far

- add x10, x10, x3
- sub x10, x10, x4
- addi x3,x4,-10

- lw x10,12(x13)
- sw   x10,40(x13)
- lb x10, 3(x11)

Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 8 GB on most 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

RISC-V Logical Instructions

- Useful for operating on fields of bits within a word
  - e.g., 8-bit character within a word
- 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; same as "<<" in C

Before: 0000 0000 0000 0000 0000 0000 0000 0010 (=2)

After: 0000 0000 0000 0000 0000 0000 0000 10**00** (=8)

What arithmetic effect does shift left have? Multiply by 2^k

- Shift Right Logical: `srli` is opposite shift; same as ">>" in C
  - Zero bits inserted at left of word, right bits shifted off end
  - Arithmetic effect? Divide by 2^k

Warning: 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 $= -25_{ten}$

- If execute **`srai x10,x10,4`** , the result is:

  1111 1111 1111 1111 1111 1111 1111 1110 $= -2_{ten}$

- Unfortunately, this is NOT same as dividing by 2^k
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Computer Decision Making

- Based on C language *if*-statement
- RISC-V:

  ```
  beq register1,register2,L1
  ```

  means: if <register1> == <register2> go to statement labeled L1
  else go to next statement
- **`beq`** stands for *branch if equal*
- The other instruction: **`bne`** for *branch if not equal*

Types of Branches

- **Branch** – change of control flow

- **Conditional Branch** – change control flow depending on comparison
  - `beq` and `bne`
  - Also branch if less than (`blt`) and branch if greater than or equal (`bge`)

- **Unconditional Branch** – always branch
  - jump (`j`)

Agenda of Lecture 1

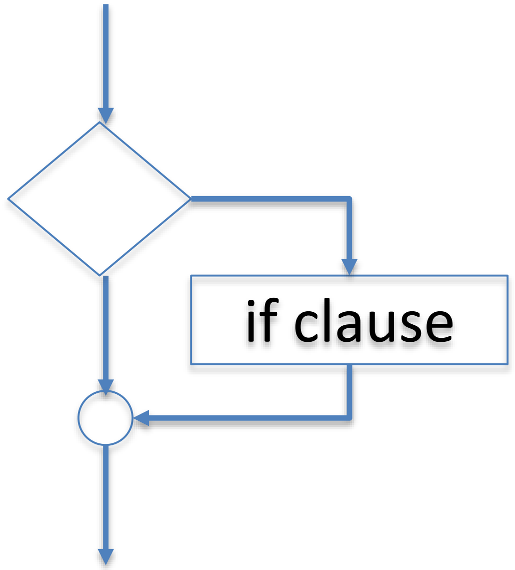
- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- Conclusion

Example *if* Statement

f	x10
g	x11
h	x12
i	x13
j	X14

- Given assignment of C variables to registers, compile this *if* block into RISC-V assembly

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

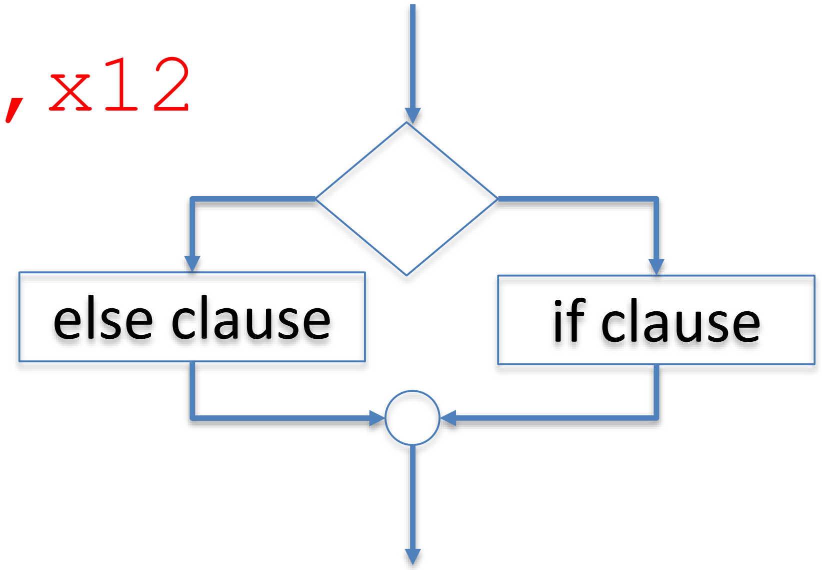


Example *if-else* Statement

f	x10
g	x11
h	x12
i	x13
j	X14

- Given same register assignments, compile

```
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
  go to label;
- "Branch on Less Than Unsigned"
  Syntax: `bltu reg1,reg2,label`
  Meaning: if (<reg1> "<" <reg2>) // treat registers as unsigned integers
  go to 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:
  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
```

Agenda of Lecture 1

- Computers
- Great Ideas in Computer Architecture
- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- 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*

Review Quiz

Which of the following is TRUE?

RED: **`add x10,x11,4(x12)`** is valid in RV32

GREEN: can byte-address 8GB of memory with an RV32 word

ORANGE: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid

YELLOW: None of the above

Review Quiz

Which of the following is TRUE?

- RED: **`add x10,x11,4(x12)`** is valid in RV32
- GREEN: can byte-address 8GB of memory with an RV32 word
- ORANGE: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid
- $\boxed{\text{YELLOW: None of the above}}$

Break!