Lecture 3 – Microarchitecture-Basics

(ECE M116C- CS M151B) Computer Architecture Systems

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <`$_s$` reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=`$_s$` reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <`$_u$` reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=`$_u$` reg[rs2]) ? label: pc + 4`
LB	`lb rd, offset(rs1)`	Load Byte	`reg[rd] <= signExtend(mem[addr])`
LH	`lh rd, offset(rs1)`	Load Half Word	`reg[rd] <= signExtend(mem[addr + 1: addr])`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[addr + 3: addr]`
LBU	`lbu rd, offset(rs1)`	Load Byte (Unsigned)	`reg[rd] <= zeroExtend(mem[addr])`
LHU	`lhu rd, offset(rs1)`	Load Half Word (Unsigned)	`reg[rd] <= zeroExtend(mem[addr + 1: addr])`
SB	`sb rs2, offset(rs1)`	Store Byte	`mem[addr] <= reg[rs2][7:0]`
SH	`sh rs2, offset(rs1)`	Store Half Word	`mem[addr + 1: addr] <= reg[rs2][15:0]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[addr + 3: addr] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <`$_s$` constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <`$_u$` constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »`$_u$` shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »`$_s$` shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <`$_s$` reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <`$_u$` reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »`$_u$` reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »`$_s$` reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

We have our assembly code, now what?

How does hardware understand these instructions?

- Machine don't understand assembly, so computers use a specific tool called *assembler* to generate machine code (ones and zeroes).

- To generate the machine code, the assembler uses a table.

How does hardware understand these instructions?

- To generate the machine code, the assembler uses a table.

0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

From Assembly to Machine Code

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

RV32I Base Instruction Set (MIT 6.004 subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Immediates

- Different instructions use different size for the immediate

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Funct3 and Funct7

- To preserve modularity

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

How to build an ISA?

- Transition from HLL to assembly
- No discussion (yet) on how
  a. to generate machine code?
  b. to upload this to hardware?
  c. does hardware run this?

How to execute these codes?

- Store-Program computers!

Store-Program Computers

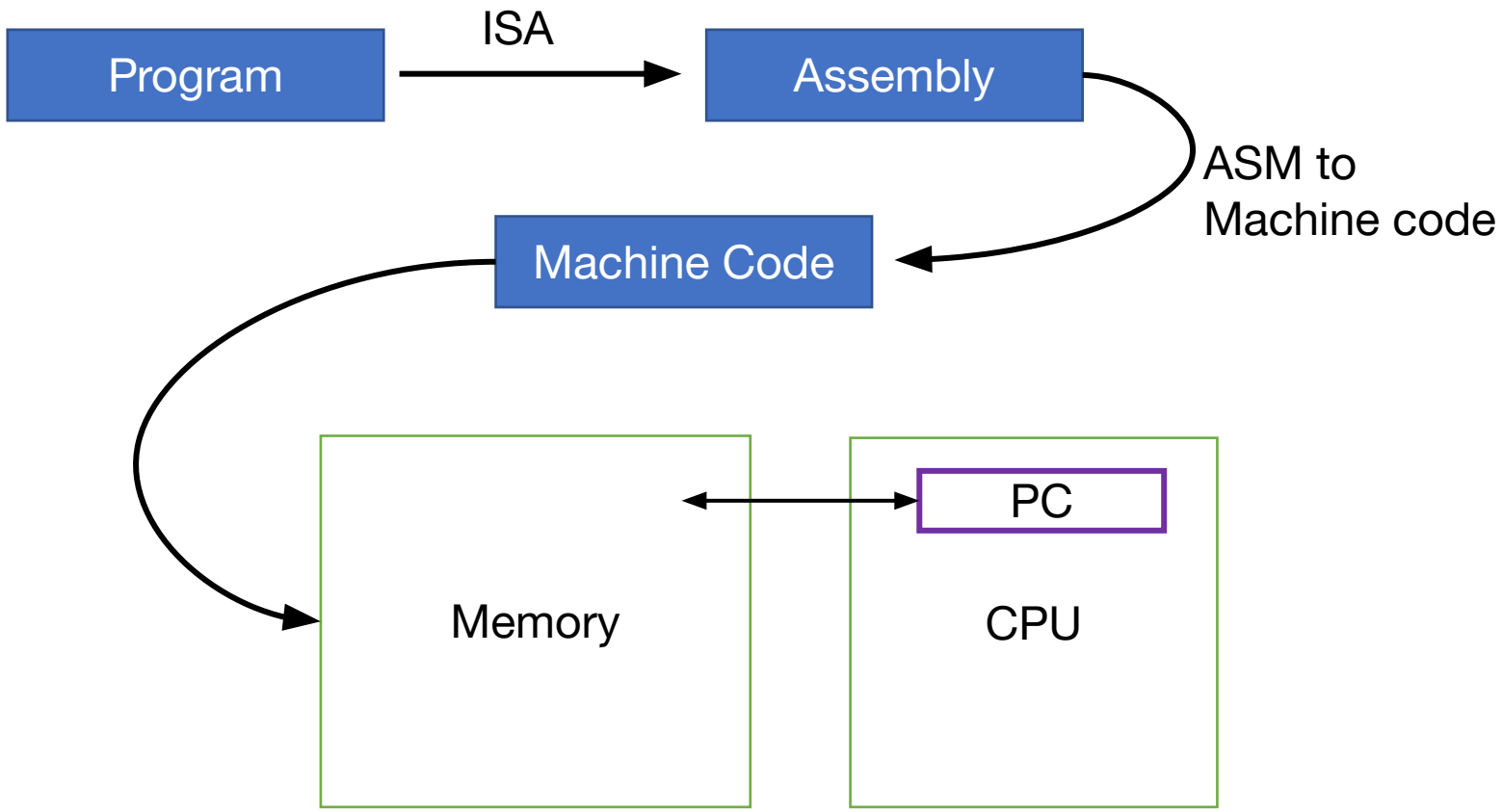
- To run programs on the processor, we store programs on the memory (same as data).

- Processor reads one instruction, updates some states and/or some memory contents, then reads the next instruction, …

- This model called Store-Program (or von-Neuman model). (there are other alternatives too!)

Executing Instruction on HW

(von Neuman model)



How to build an ISA?

- Transition from HLL to assembly
- No discussion (yet) on how
  a. to generate machine code?
  b. to upload this to hardware?
  c. does hardware run this?

How to implement a given ISA?

Microarchitecture

- Hardware implementation of ISA is called *microarchitecture*.

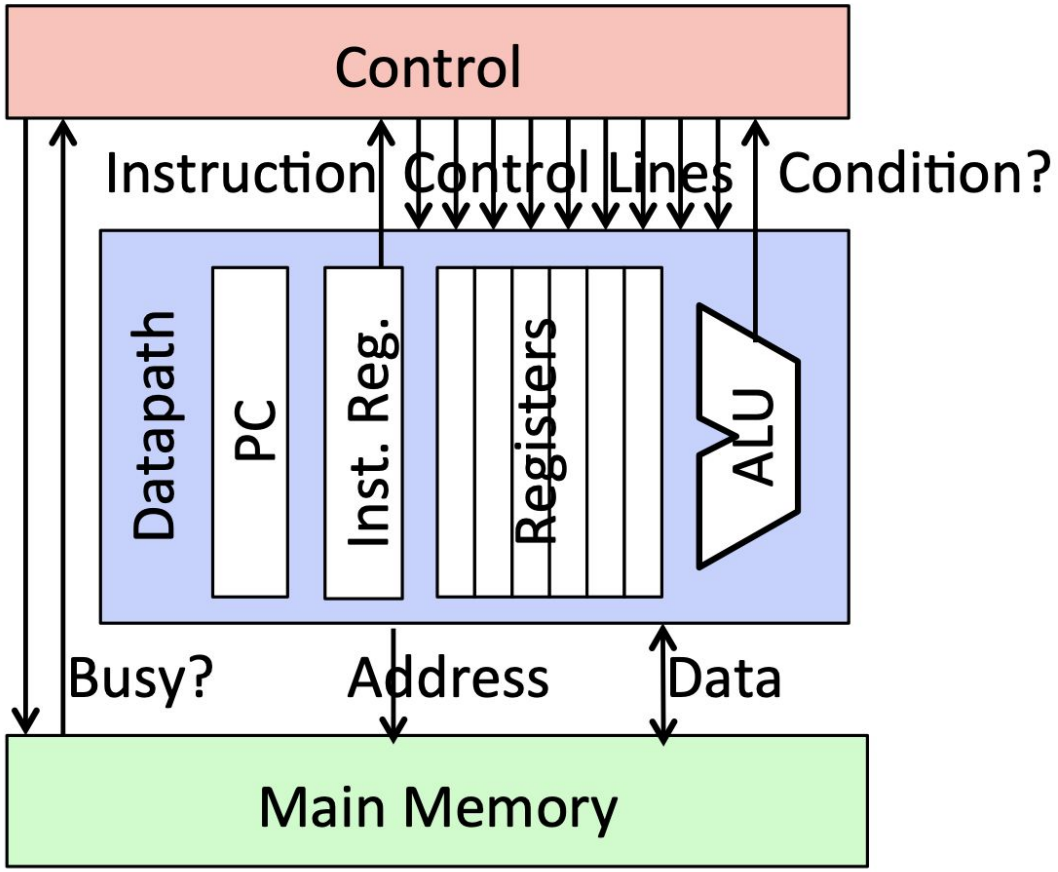
Microarchitecture

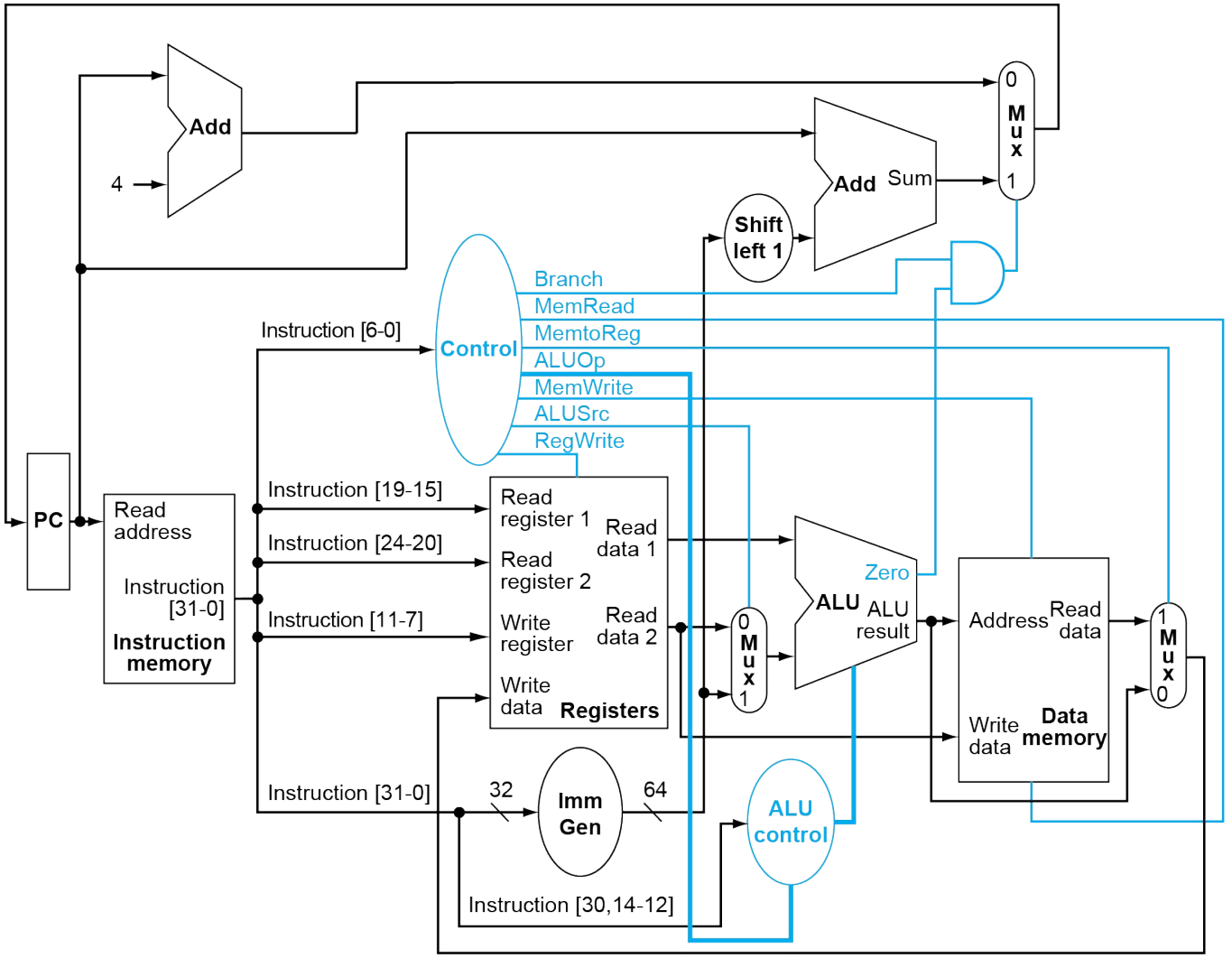
- Hardware implementation of ISA is called *microarchitecture*.
- A given ISA (e.g., RISC-V) can be implemented in many different ways (i.e., same ISA, different microarchitecture).
- The goal is to build an *efficient* computer.
  - Fast (high performance) and power-efficient.

What strategy to use?

State-Machine View

State-Machine View





*Images were taken from Hennessy Patterson Book [1].

Datapath + Controller

Lifecycle of an instruction

Lifecycle of an instruction

1- Instruction is read (fetch) from the (instruction) memory

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Lifecycle of an instruction

1- Instruction is read (fetch) from the (instruction) memory

2- Operands should be loaded
- RegFile, (data) Memory, Imm.
  - Need register number/ address for each operand.

Lifecycle of an instruction

1- Instruction is read (fetch) from the (instruction) memory

2- Operands should be loaded
- RegFile, (data) Memory, Imm.
  - Need register number/ address for each operand.

3- Operation should be executed
- Arithmetic (which type?), data movement, control-flow.

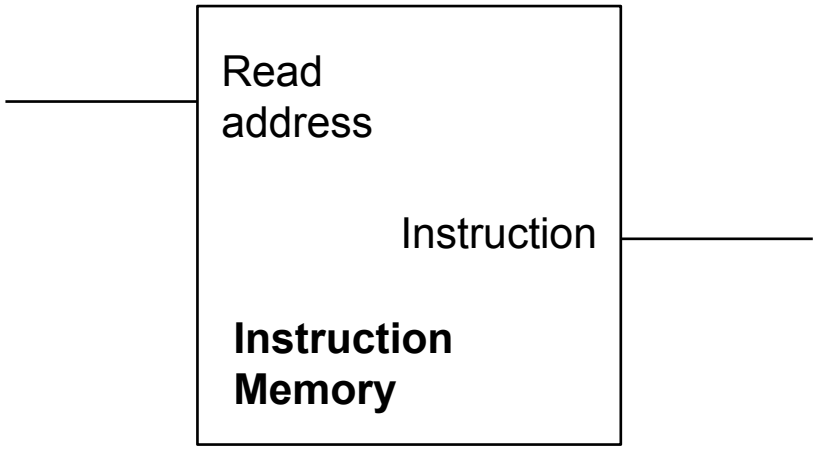
Lifecycle of an instruction

1- Instruction is read (fetch) from the (instruction) memory
2- Operands should be loaded
   - RegFile, (data) Memory, Imm.
     - Need register number/ address for each operand.
3- Operation should be executed
   - Arithmetic (which type?), data movement, control-flow
4- Results should be stored
   - RegFile or memory?

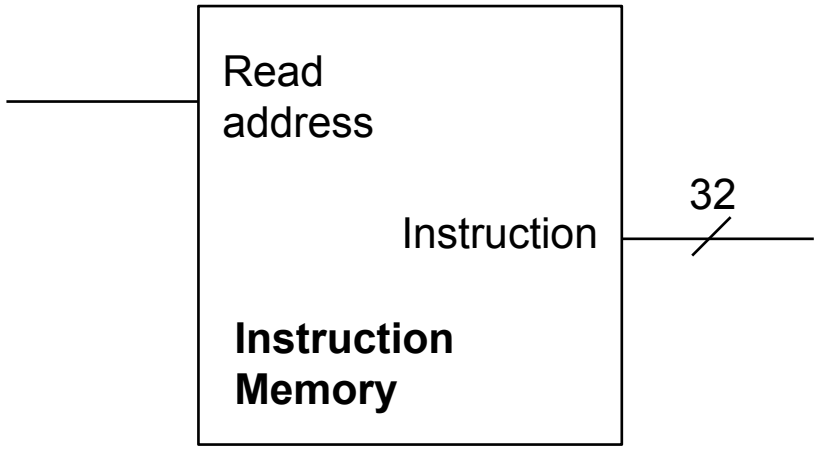
Lifecycle of an instruction

1- Instruction is read (fetch) from the (instruction) memory
2- Operands should be loaded
   - RegFile, (data) Memory, Imm.
     - Need register number/ address for each operand.
3- Operation should be executed
   - Arithmetic (which type?), data movement, control-flow.
4- Results should be stored
   - RegFile or memory?
5- PC should be updated
   - Sequential, jump, or branch?

Instruction Fetch



Instruction Fetch



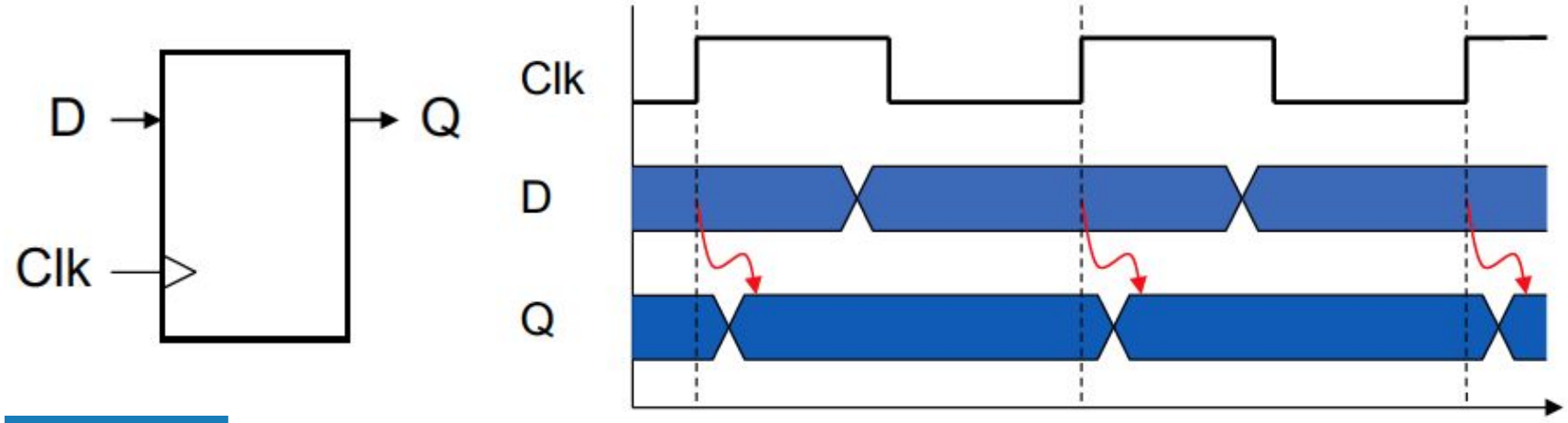
Umm, how reading from memory (sequential logic) was working?

Logic Design Detour

- Basics
  - Combinational element
    - Operate on data
    - Output is a function of input
  - State (sequential) elements
    - Store information

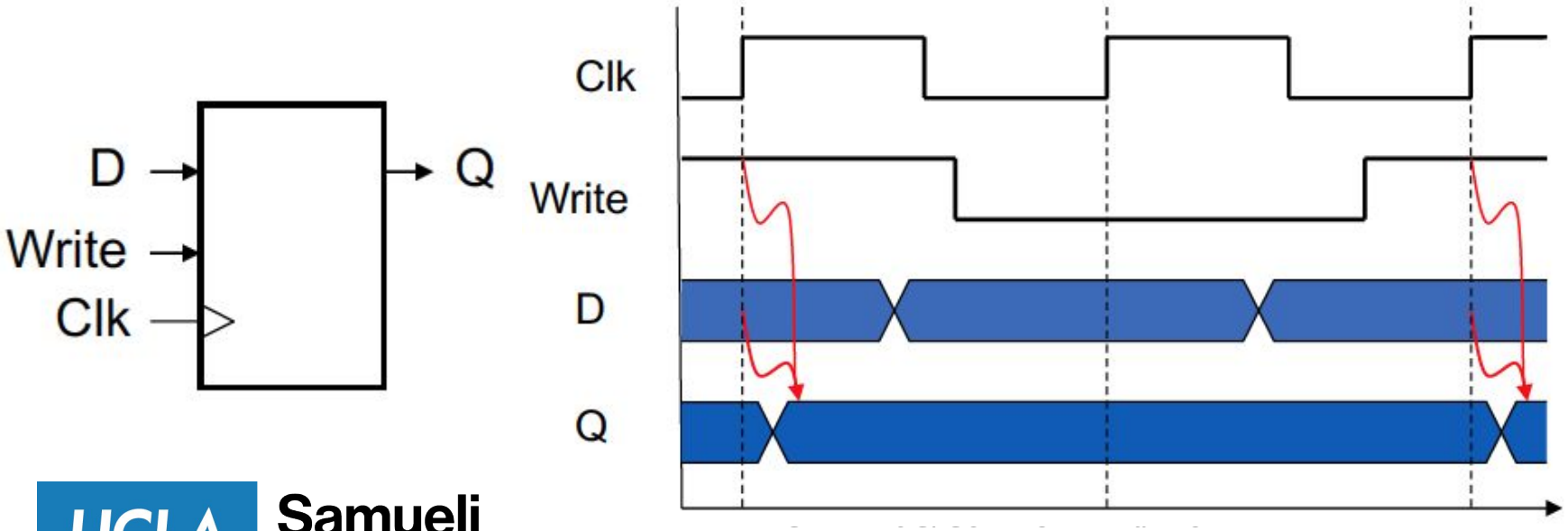
Sequential Logic

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value (could be multiple bits).
  - (Positive) *Edge-triggered:* update when *Clk* changes from 0 to 1.



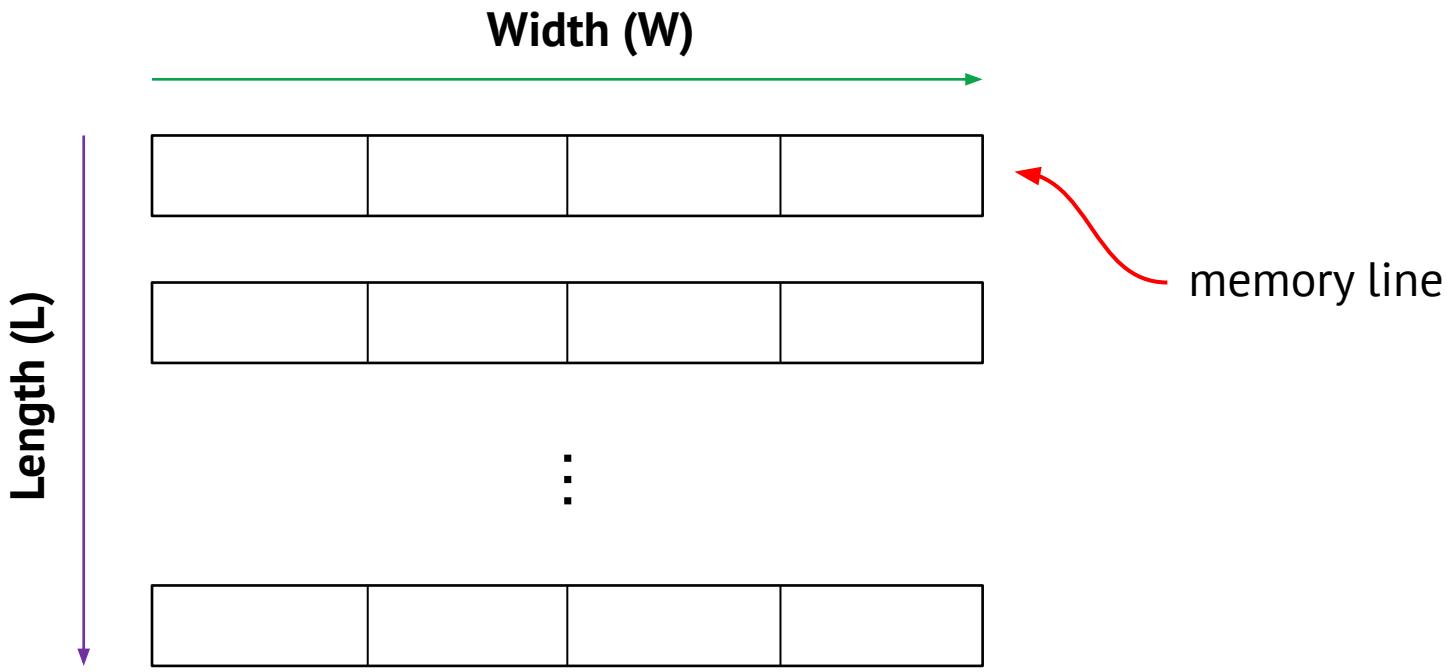
Sequential Logic

- Register with *write enable*
  - Only updates on clock edge when write control input is 1 (called active high write enable).
  - Used when only sometimes we are allowed to write.



Sequential Logic

- Memory – array of *registers**



W=? L = ?

Sequential Logic

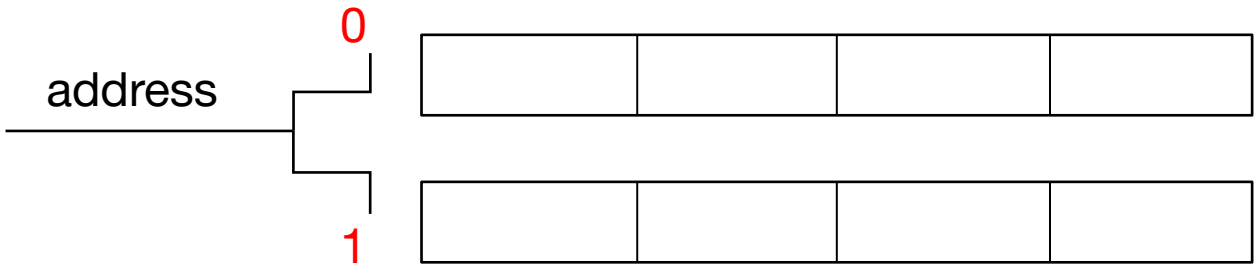
- Memory – array of *registers**

  – *How to choose one line?*

Sequential Logic

- Memory – array of *registers**

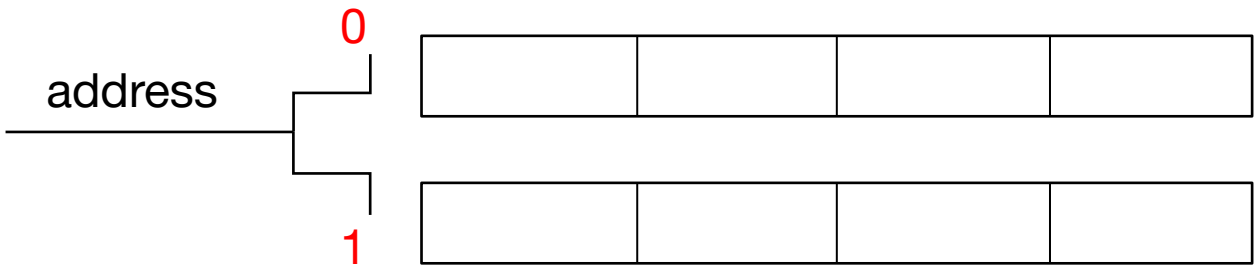
  – *How to choose one line?*



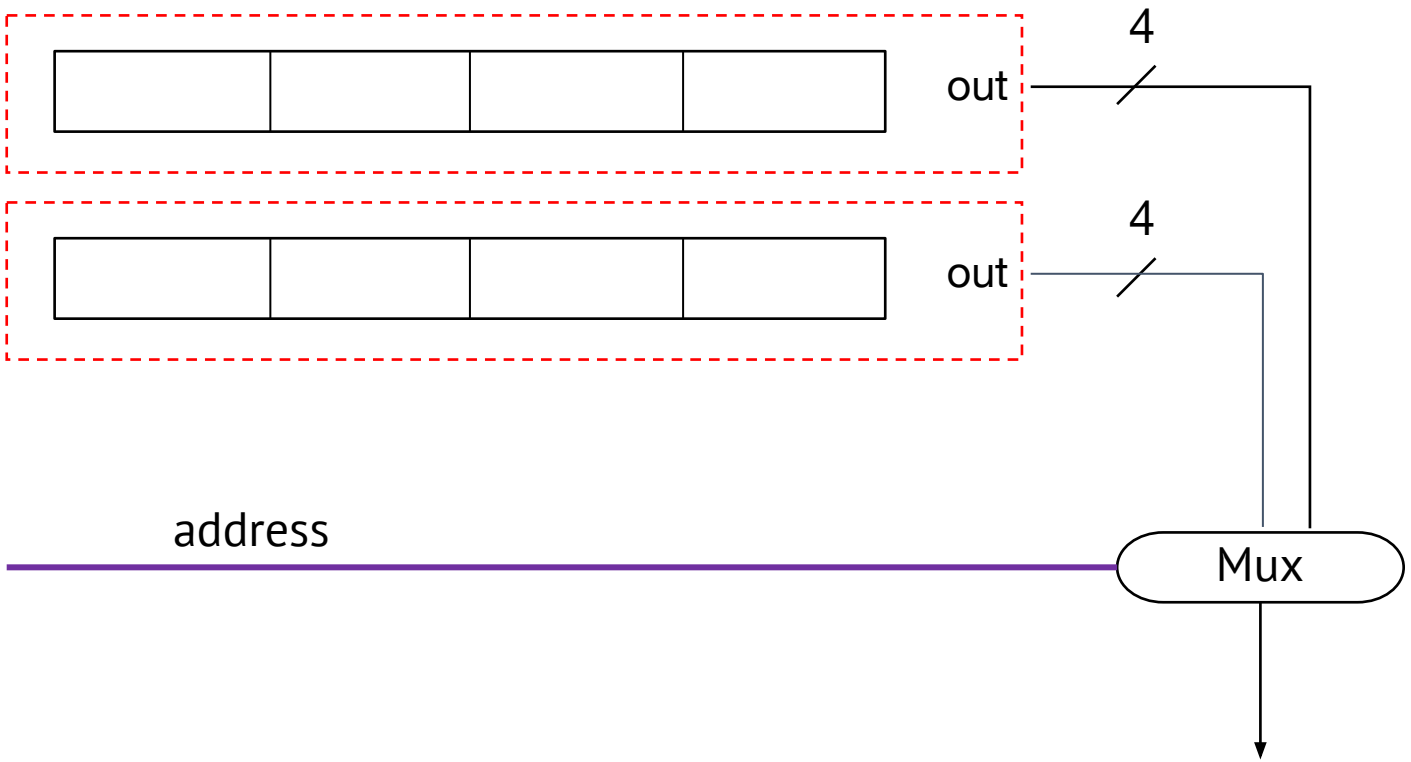
Sequential Logic

- Memory – array of *registers**

  – *How to choose one line?* ***Multiplexer!***



Reading a Memory Line



- read/write enable
- more lines

Memory Technology

- Are all memory cells registers, SRAM, DRAM, etc.? *(why?)*

Different Storage (Memory) Elements

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very expensive (one bit costs ***tens*** of transistors).

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Different Storage (Memory) Elements

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very expensive (one bit costs ***tens*** of transistors).
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time.
  - Expensive (one bit costs **6+** transistors).
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing.
  - Cheap (one bit costs only ***one*** transistor plus one capacitor).

Different Storage (Memory) Elements

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very expensive (one bit costs ***tens*** of transistors).
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time.
  - Expensive (one bit costs **6+** transistors).
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing.
  - Cheap (one bit costs only ***one*** transistor plus one capacitor).
- DISK (*flash memory*, *hard disk*)
  - Much slower, access takes a long time, ***non-volatile.***
  - Very cheap.

Which storage element to use?

- **Trend** (Register to DRAM to Disk):
  - From *smaller* and *faster* →*bigger* and *slower*
  - From *more expensive* and *power hungry* →*cheaper* and *power efficient*

Which storage element to use?

- **Trend** (Register to DRAM to Disk):
  - From *smaller* and *faster* →*bigger* and *slower*
  - From *more expensive* and *power hungry* →*cheaper* and *power efficient*

➔ ***IDEA:*** **Memory Hierarchy**

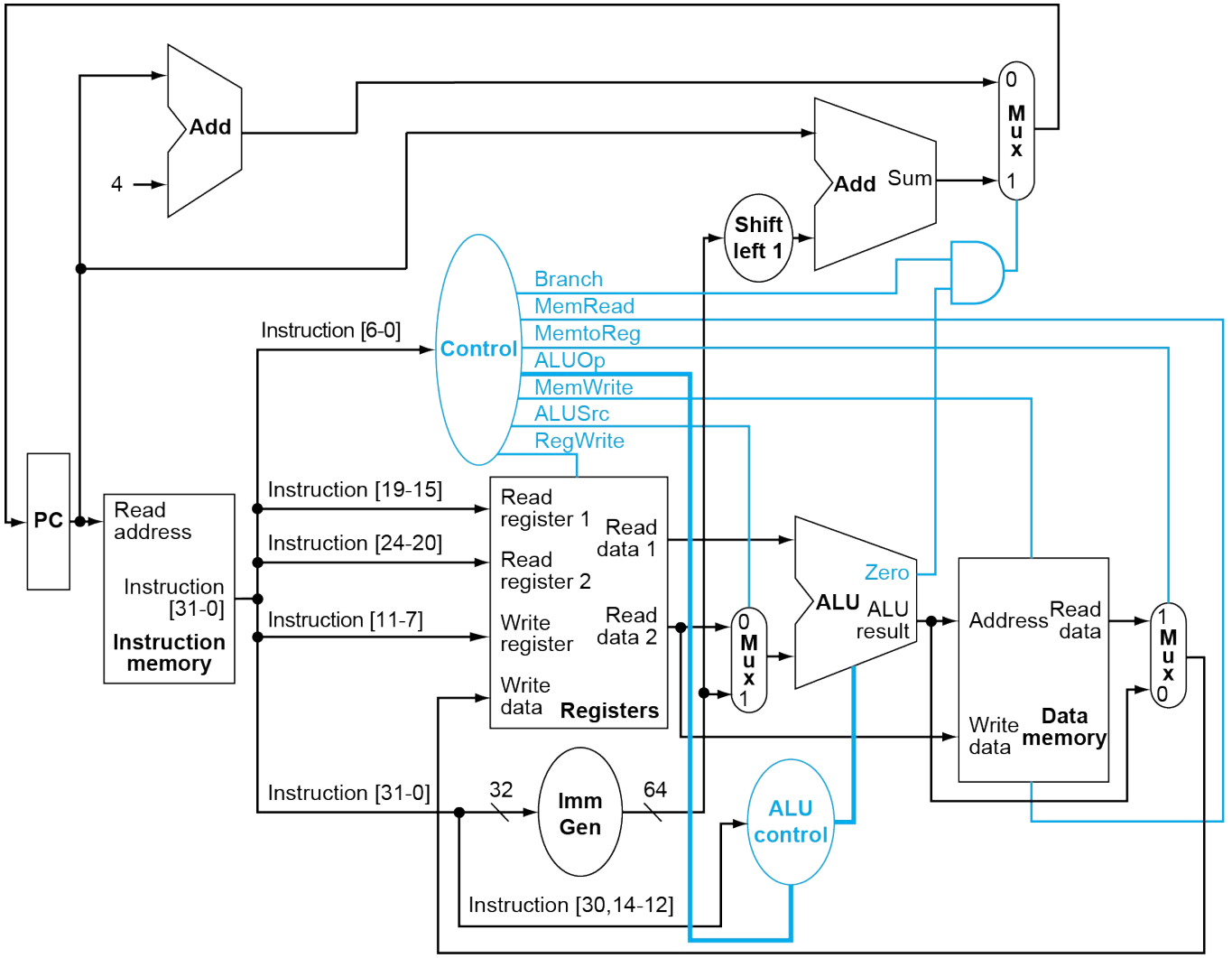
- Store data in *different layers*. Use *smaller* and *faster* elements ***closer*** to the processor to store ***frequently-used*** data. Use *slower* but *bigger* elements to store *rarely-used* permanent data.

Which storage element to use?

- **Trend** (Register to DRAM to Disk):
  - From *smaller* and *faster* →*bigger* and *slower*
  - From *more expensive* and *power hungry* →*cheaper* and *P*

We will talk more about this soon!

➔
- Store data in *different layers*. Use *smaller* and *faster* elements ***closer*** to the processor to store ***frequently-used*** data. Use *slower* but *bigger* elements to store *rarely-used* permanent data.



*Images were taken from Hennessy Patterson Book [1].

Datapath + Controller

Participation

- Link: https://gavel-for-nader.web.app/lo
  or Scan the QR code.

- Use this password: [redacted]

Summary

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Goergia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)