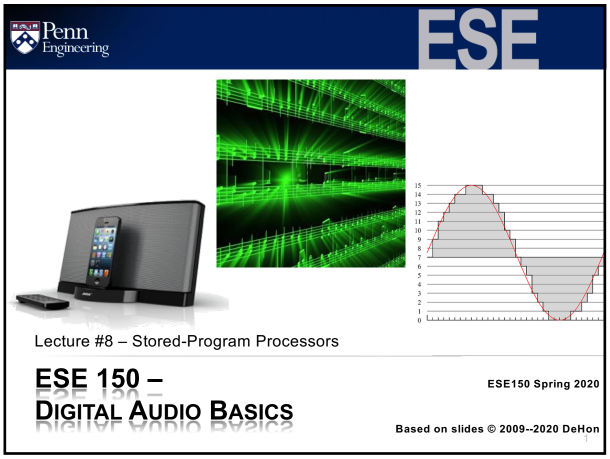
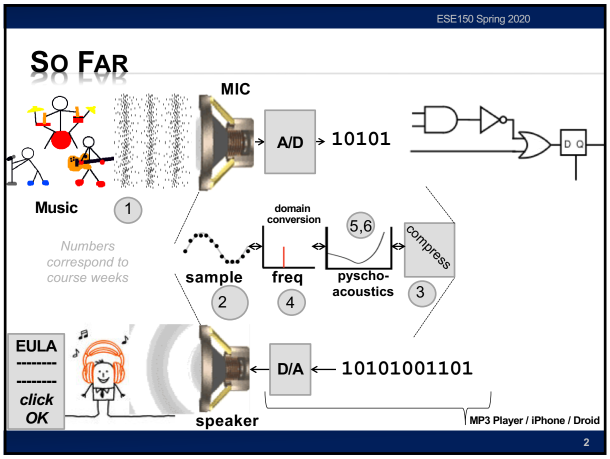

Penn Engineering
ESE

Lecture #8 – Stored-Program Processors

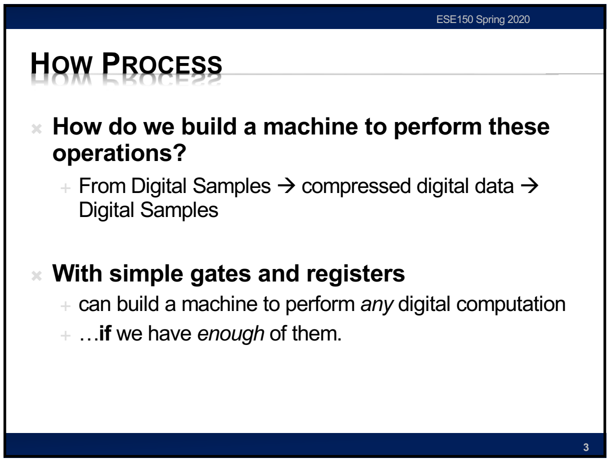
## ESE 150 – Digital Audio Basics

ESE150 Spring 2020

Based on slides © 2009--2020 DeHon

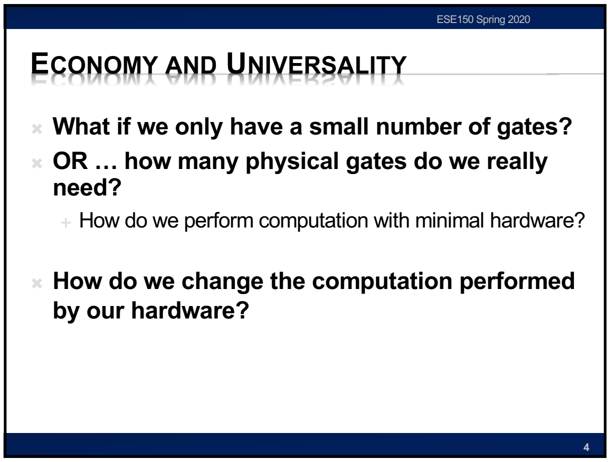


## So Far



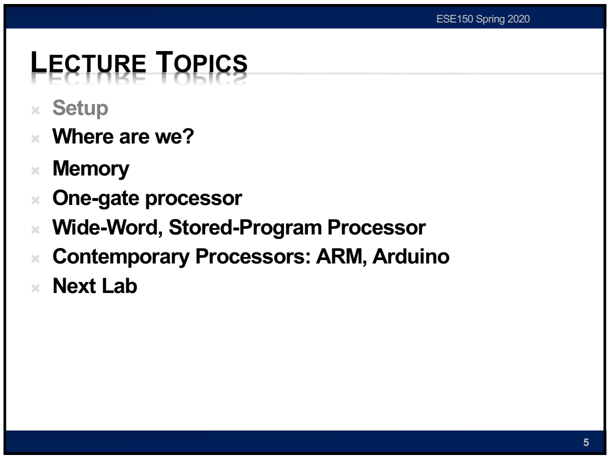
## How Process

- **How do we build a machine to perform these operations?**
  - From Digital Samples → compressed digital data → Digital Samples
- **With simple gates and registers**
  - can build a machine to perform *any* digital computation
  - …**if** we have *enough* of them.



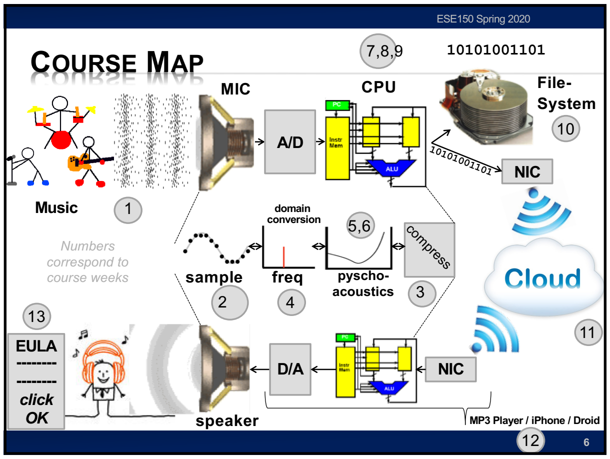
## Economy and Universality

- **What if we only have a small number of gates?**
- **OR … how many physical gates do we really need?**
  - How do we perform computation with minimal hardware?
- **How do we change the computation performed by our hardware?**



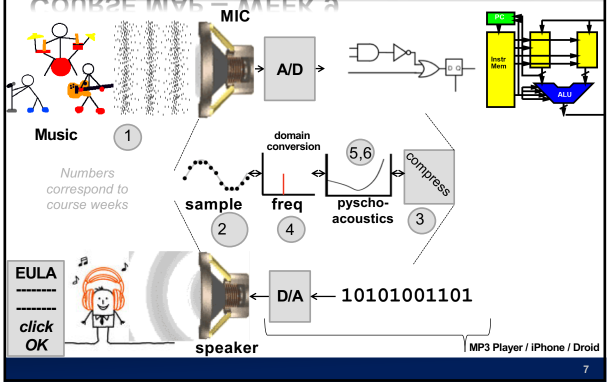
## Lecture Topics

- Setup
- **Where are we?**
- **Memory**
- **One-gate processor**
- **Wide-Word, Stored-Program Processor**
- **Contemporary Processors: ARM, Arduino**
- **Next Lab**



## Course Map

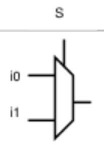
## Course Map – Week 9



## Quick Reminder

## Multiplexer Gate

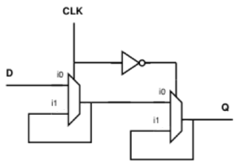
- **MUX**
  - When S=0, output=i0
  - When S=1, output=i1

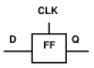


| S | i0 | i1 | Mux2(S,i0,i1) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

## State Element

- **Latch or Register is a state element**
- **Allows circuit to *remember* a value**
- **Build computations that**
  - Depend on past inputs
  - Reuse hardware in time

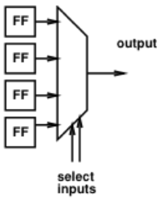




## Mux can be a programmable Gate

- **Programmable Gate**
  - Can be programmed to act as any gate
  - Use state (e.g. FF) to “program” truth table of a gate

| Input 0 | Input 1 | Output |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |



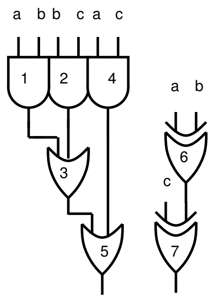
## NAND Universality

- **Can implement any combinational logic function out of a collection of NAND2 gates**
  - Or AND, OR, NOT combination
  - Or Programmable MUX gates (OR)

## Preclass 1

- **What Function?**
  - o1=a&b | b&c | a&c;
  - o2=a^b^c;
- **How many gates?**

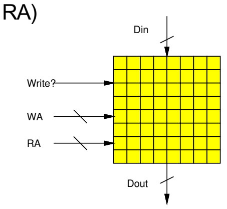
## Preclass 1 in Gates



## Memory

## Random Access Memory

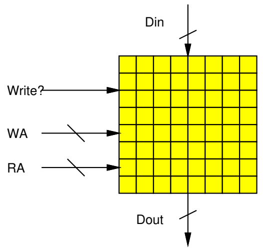
- **A Memory:**
  - Series of locations (slots)
  - Can write values a slot (specified by address, WA)
  - Read values from (by address, RA)
  - Return last value written



Notation:
slash on wire
means multiple bits wide

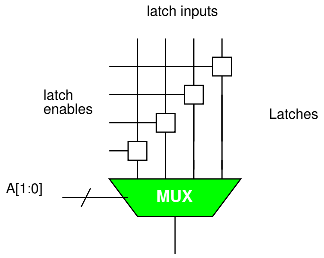
## Two Pieces of a Memory

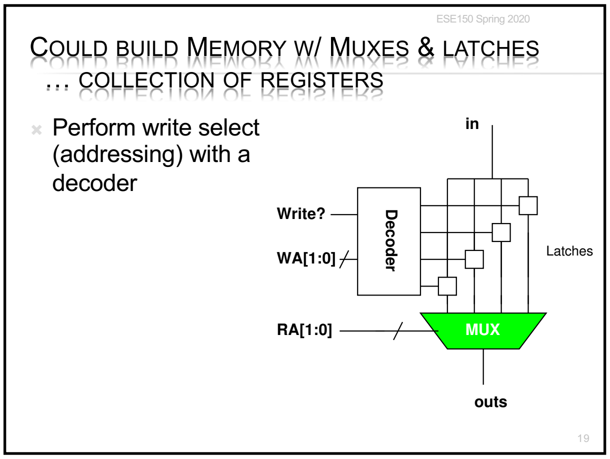
1. **Element to remember a value**
2. **Way to address/select that element**



## Could build Memory w/ Muxes & Latches … collection of registers

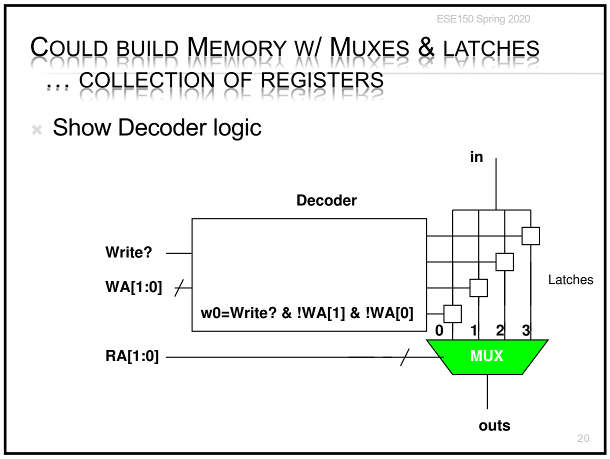
- Use latch to remember (store) values
- Perform read select (addressing) with a multiplexer





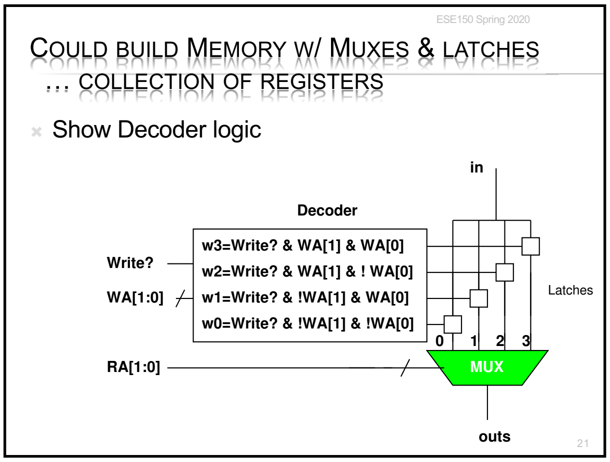
## Could build Memory w/ Muxes & Latches … collection of registers

- Perform write select (addressing) with a decoder



## Could build Memory w/ Muxes & Latches … collection of registers

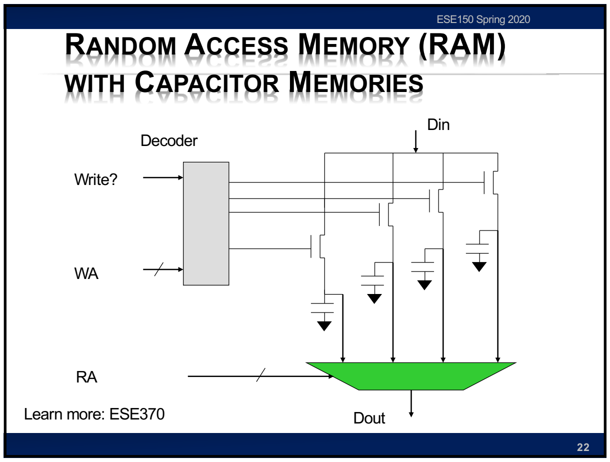
- Show Decoder logic



## Could build Memory w/ Muxes & Latches … collection of registers

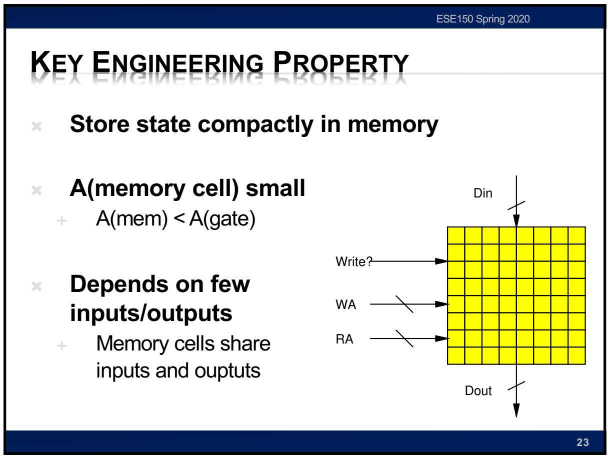
- Show Decoder logic

in
Decoder
w3=Write? & WA[1] & WA[0]
w2=Write? & WA[1] & ! WA[0]
w1=Write? & !WA[1] & WA[0]
w0=Write? & !WA[1] & !WA[0]
Write?
WA[1:0]
Latches
0
1
2
3
RA[1:0]
MUX
outs



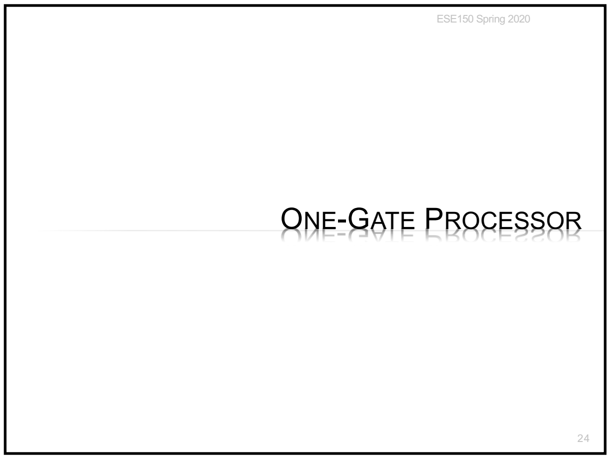
## Random Access Memory (RAM) with Capacitor Memories

Learn more: ESE370

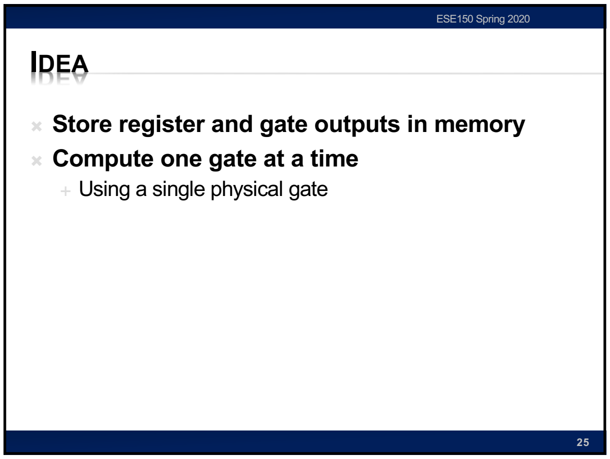


## Key Engineering Property

- **Store state compactly in memory**
- **A(memory cell) small**
  - A(mem) < A(gate)
- **Depends on few inputs/outputs**
  - Memory cells share inputs and ouptuts

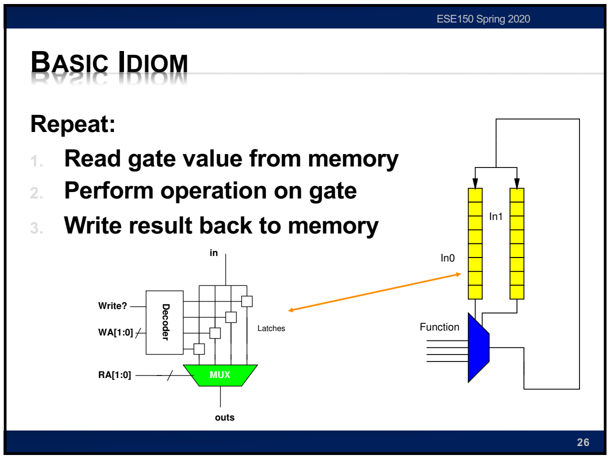


## One-Gate Processor



## Idea

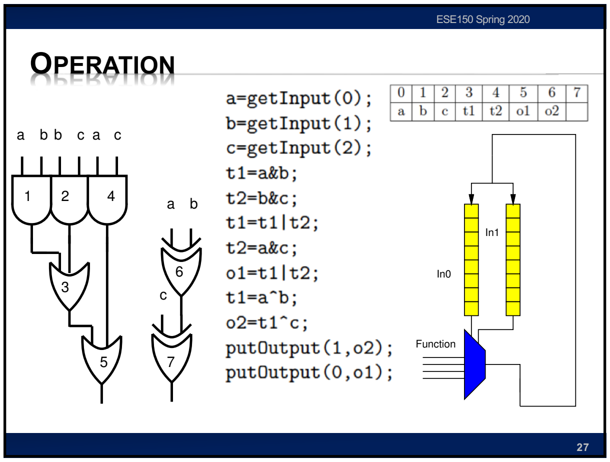
- Store register and gate outputs in memory
- Compute one gate at a time
  - Using a single physical gate



## Basic Idiom

Repeat:

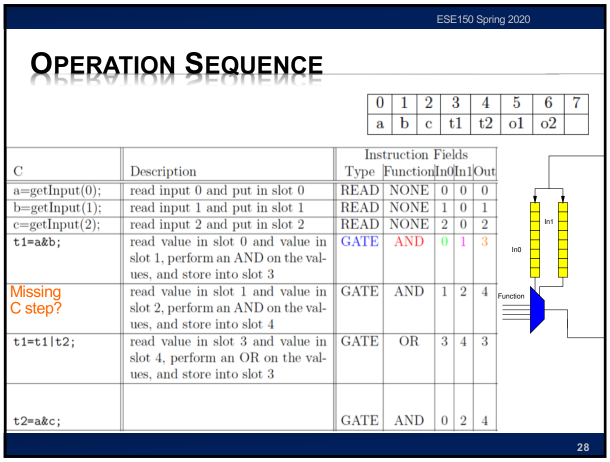
1. Read gate value from memory
2. Perform operation on gate
3. Write result back to memory



## Operation

```
a=getInput(0);
b=getInput(1);
c=getInput(2);
t1=a&b;
t2=b&c;
t1=t1|t2;
t2=a&c;
o1=t1|t2;
t1=a^b;
o2=t1^c;
putOutput(1,o2);
putOutput(0,o1);
```

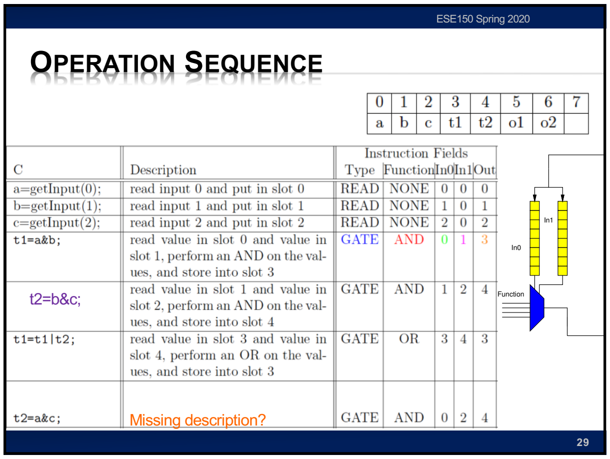
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 | |



## Operation Sequence

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 | |

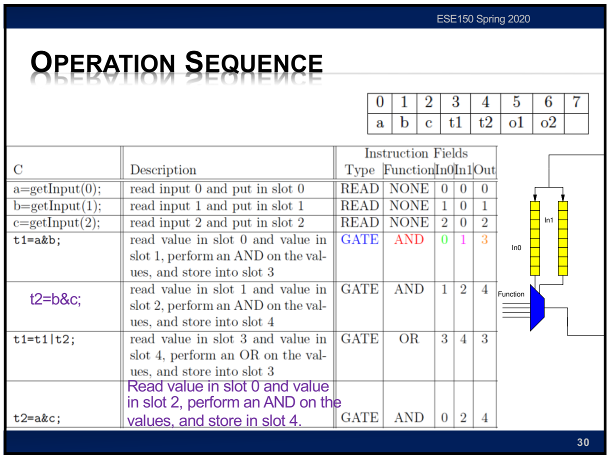
| | | Instruction Fields | | | | |
|---|---|---|---|---|---|---|
| C | Description | Type | Function | In0 | In1 | Out |
| a=getInput(0); | read input 0 and put in slot 0 | READ | NONE | 0 | 0 | 0 |
| b=getInput(1); | read input 1 and put in slot 1 | READ | NONE | 1 | 0 | 1 |
| c=getInput(2); | read input 2 and put in slot 2 | READ | NONE | 2 | 0 | 2 |
| t1=a&b; | read value in slot 0 and value in slot 1, perform an AND on the values, and store into slot 3 | GATE | AND | 0 | 1 | 3 |
| Missing C step? | read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 4 | GATE | AND | 1 | 2 | 4 |
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 |
| t2=a&c; | | GATE | AND | 0 | 2 | 4 |



## Operation Sequence

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 | |

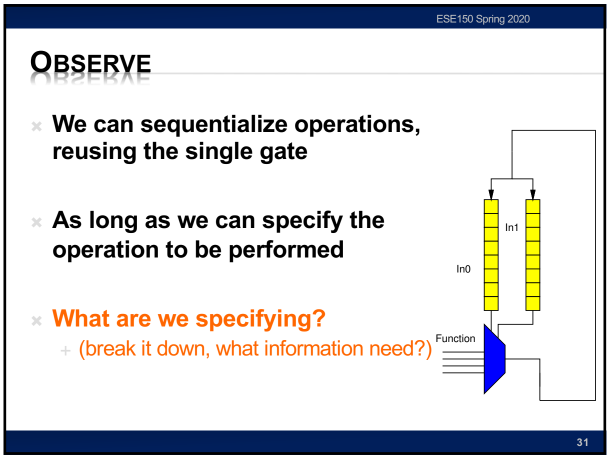
| | | Instruction Fields | | | | |
|---|---|---|---|---|---|---|
| C | Description | Type | Function | In0 | In1 | Out |
| a=getInput(0); | read input 0 and put in slot 0 | READ | NONE | 0 | 0 | 0 |
| b=getInput(1); | read input 1 and put in slot 1 | READ | NONE | 1 | 0 | 1 |
| c=getInput(2); | read input 2 and put in slot 2 | READ | NONE | 2 | 0 | 2 |
| t1=a&b; | read value in slot 0 and value in slot 1, perform an AND on the values, and store into slot 3 | GATE | AND | 0 | 1 | 3 |
| t2=b&c; | read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 4 | GATE | AND | 1 | 2 | 4 |
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 |
| t2=a&c; | Missing description? | GATE | AND | 0 | 2 | 4 |



## Operation Sequence

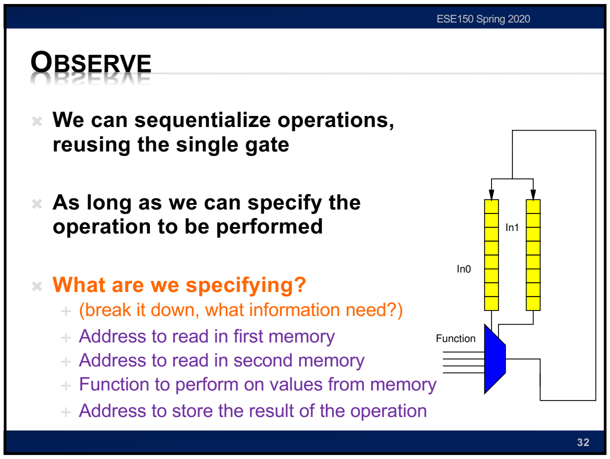
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 | |

| | | Instruction Fields | | | | |
|---|---|---|---|---|---|---|
| C | Description | Type | Function | In0 | In1 | Out |
| a=getInput(0); | read input 0 and put in slot 0 | READ | NONE | 0 | 0 | 0 |
| b=getInput(1); | read input 1 and put in slot 1 | READ | NONE | 1 | 0 | 1 |
| c=getInput(2); | read input 2 and put in slot 2 | READ | NONE | 2 | 0 | 2 |
| t1=a&b; | read value in slot 0 and value in slot 1, perform an AND on the values, and store into slot 3 | GATE | AND | 0 | 1 | 3 |
| t2=b&c; | read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 4 | GATE | AND | 1 | 2 | 4 |
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 |
| t2=a&c; | Read value in slot 0 and value in slot 2, perform an AND on the values, and store in slot 4. | GATE | AND | 0 | 2 | 4 |



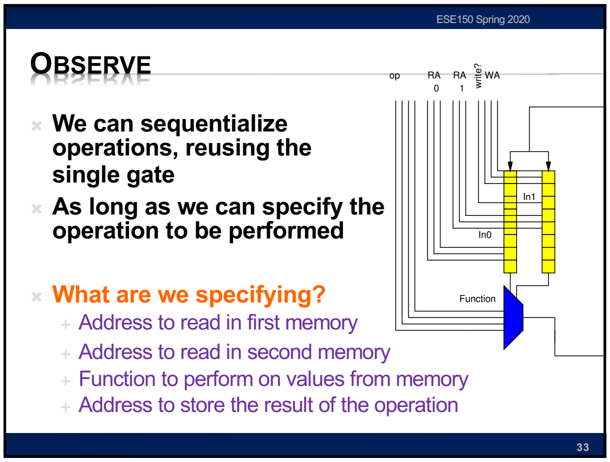
## Observe

- We can sequentialize operations, reusing the single gate
- As long as we can specify the operation to be performed
- What are we specifying?
  - (break it down, what information need?)



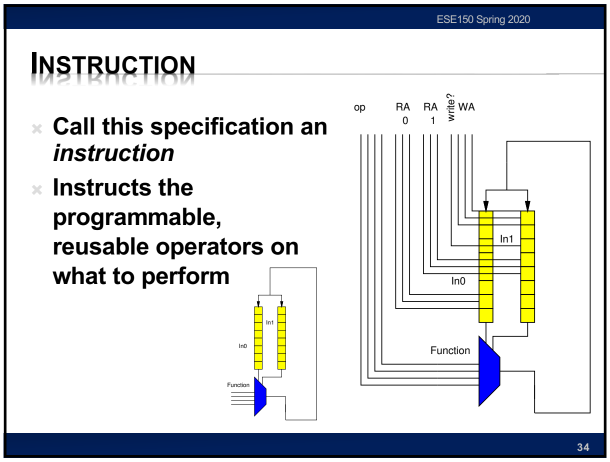
## Observe

- We can sequentialize operations, reusing the single gate
- As long as we can specify the operation to be performed
- What are we specifying?
  - (break it down, what information need?)
  - Address to read in first memory
  - Address to read in second memory
  - Function to perform on values from memory
  - Address to store the result of the operation



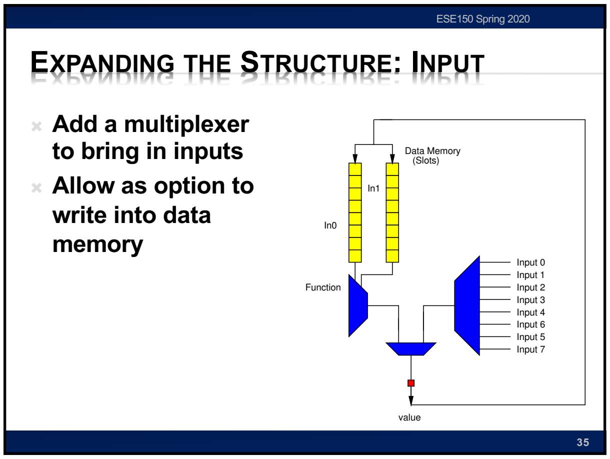
## Observe

- We can sequentialize operations, reusing the single gate
- As long as we can specify the operation to be performed
- What are we specifying?
  - Address to read in first memory
  - Address to read in second memory
  - Function to perform on values from memory
  - Address to store the result of the operation



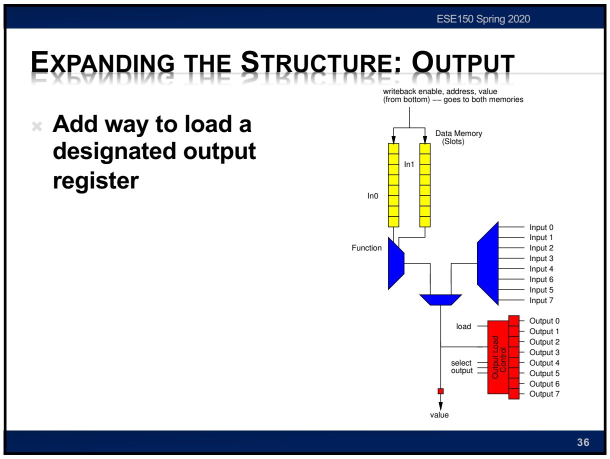
## Instruction

- Call this specification an *instruction*
- Instructs the programmable, reusable operators on what to perform



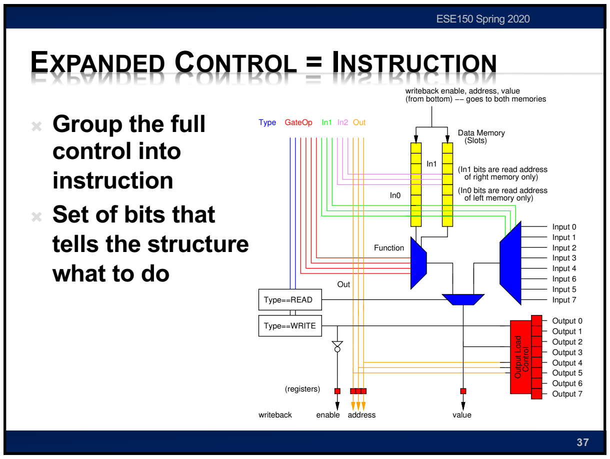
## Expanding the Structure: Input

- Add a multiplexer to bring in inputs
- Allow as option to write into data memory



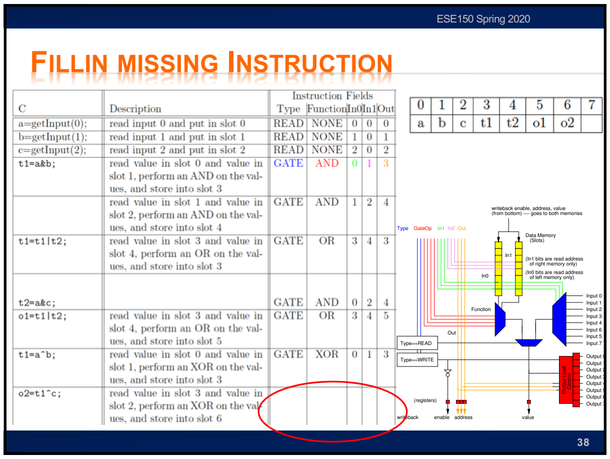
## Expanding the Structure: Output

- Add way to load a designated output register



## Expanded Control = Instruction

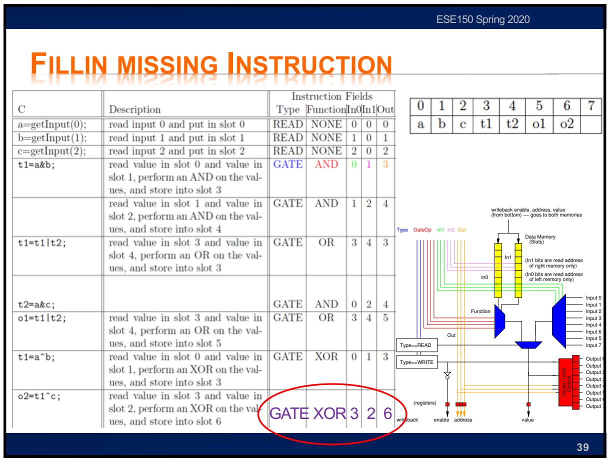
- Group the full control into instruction
- Set of bits that tells the structure what to do



## Fillin missing Instruction

| C | Description | Instruction Fields<br>Type | Function | In0 | In1 | Out |
|---|---|---|---|---|---|---|
| a=getInput(0); | read input 0 and put in slot 0 | READ | NONE | 0 | 0 | 0 |
| b=getInput(1); | read input 1 and put in slot 1 | READ | NONE | 1 | 0 | 1 |
| c=getInput(2); | read input 2 and put in slot 2 | READ | NONE | 2 | 0 | 2 |
| t1=a&b; | read value in slot 0 and value in slot 1, perform an AND on the values, and store into slot 3 | GATE | AND | 0 | 1 | 3 |
|  | read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 4 | GATE | AND | 1 | 2 | 4 |
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 |
| t2=a&c; |  | GATE | AND | 0 | 2 | 4 |
| o1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 5 | GATE | OR | 3 | 4 | 5 |
| t1=a^b; | read value in slot 0 and value in slot 1, perform an XOR on the values, and store into slot 3 | GATE | XOR | 0 | 1 | 3 |
| o2=t1^c; | read value in slot 3 and value in slot 2, perform an XOR on the values, and store into slot 6 |  |  |  |  |  |

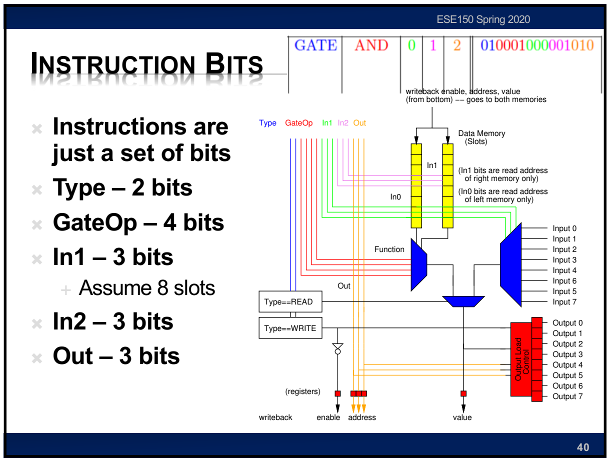
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 |  |



## Fillin missing Instruction

| C | Description | Instruction Fields<br>Type | Function | In0 | In1 | Out |
|---|---|---|---|---|---|---|
| a=getInput(0); | read input 0 and put in slot 0 | READ | NONE | 0 | 0 | 0 |
| b=getInput(1); | read input 1 and put in slot 1 | READ | NONE | 1 | 0 | 1 |
| c=getInput(2); | read input 2 and put in slot 2 | READ | NONE | 2 | 0 | 2 |
| t1=a&b; | read value in slot 0 and value in slot 1, perform an AND on the values, and store into slot 3 | GATE | AND | 0 | 1 | 3 |
|  | read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 4 | GATE | AND | 1 | 2 | 4 |
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 |
| t2=a&c; |  | GATE | AND | 0 | 2 | 4 |
| o1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 5 | GATE | OR | 3 | 4 | 5 |
| t1=a^b; | read value in slot 0 and value in slot 1, perform an XOR on the values, and store into slot 3 | GATE | XOR | 0 | 1 | 3 |
| o2=t1^c; | read value in slot 3 and value in slot 2, perform an XOR on the values, and store into slot 6 | GATE | XOR | 3 | 2 | 6 |

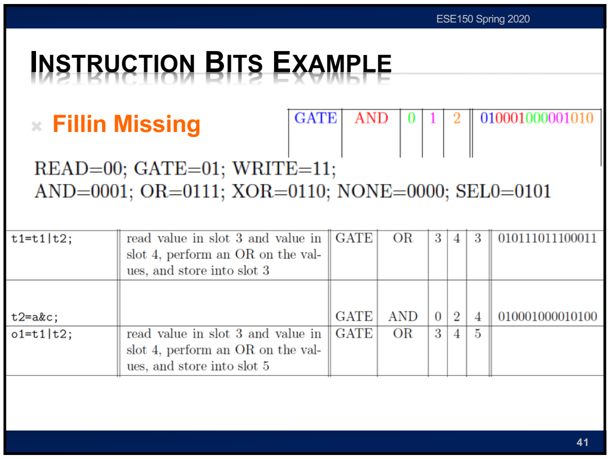
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | b | c | t1 | t2 | o1 | o2 |  |



## Instruction Bits

| GATE | AND | 0 | 1 | 2 | 010001000001010 |
|---|---|---|---|---|---|

- Instructions are just a set of bits
- Type – 2 bits
- GateOp – 4 bits
- In1 – 3 bits
  - Assume 8 slots
- In2 – 3 bits
- Out – 3 bits



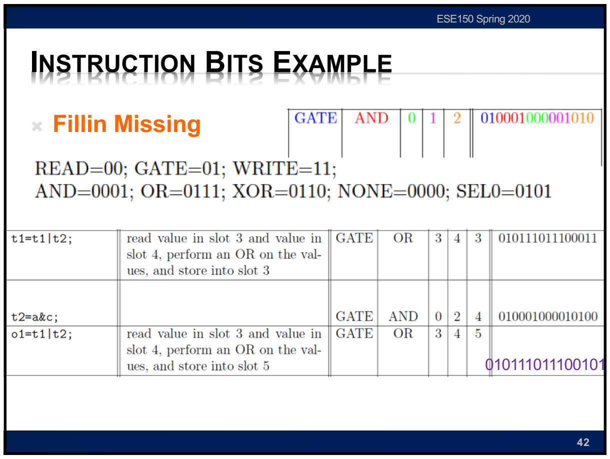
## Instruction Bits Example

- Fillin Missing

| GATE | AND | 0 | 1 | 2 | 010001000001010 |
|---|---|---|---|---|---|

READ=00; GATE=01; WRITE=11;
AND=0001; OR=0111; XOR=0110; NONE=0000; SEL0=0101

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 | 010111011100011 |
| t2=a&c; |  | GATE | AND | 0 | 2 | 4 | 010001000010100 |
| o1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 5 | GATE | OR | 3 | 4 | 5 |  |



## Instruction Bits Example

- Fillin Missing

| GATE | AND | 0 | 1 | 2 | 010001000001010 |
|---|---|---|---|---|---|

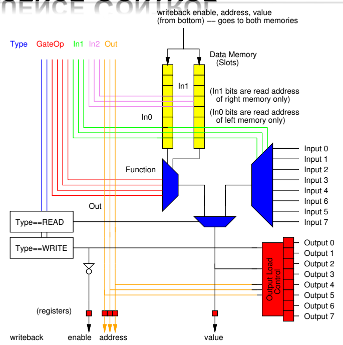
READ=00; GATE=01; WRITE=11;
AND=0001; OR=0111; XOR=0110; NONE=0000; SEL0=0101

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| t1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 3 | GATE | OR | 3 | 4 | 3 | 010111011100011 |
| t2=a&c; |  | GATE | AND | 0 | 2 | 4 | 010001000010100 |
| o1=t1\|t2; | read value in slot 3 and value in slot 4, perform an OR on the values, and store into slot 5 | GATE | OR | 3 | 4 | 5 | 010111011100101 |

ESE150 Spring 2020

# Instruction Sequence Control

- How provide the sequence of instructions?

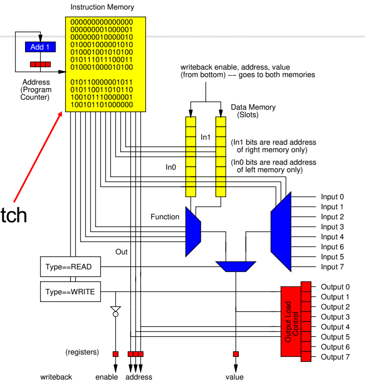


43

ESE150 Spring 2020

# Instruction Memory

- Add Memory to hold set of Instructions
  - Note contents match table on p. 2 of preclass
- Counter to sequence instructions

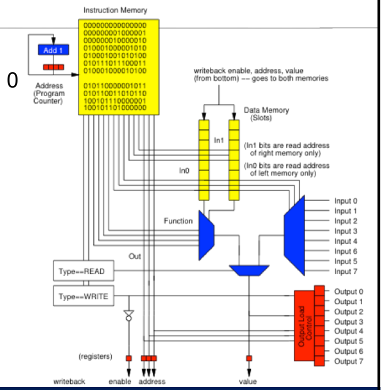


44

ESE150 Spring 2020

# Animate

- Start at PC=0

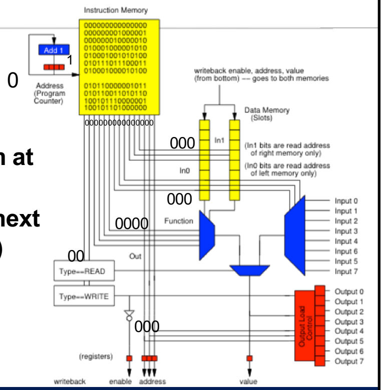


45

ESE150 Spring 2020

# Animate

- Start at PC=0
- Read Instr. Mem at 0
- (also compute next PC by adding 1)

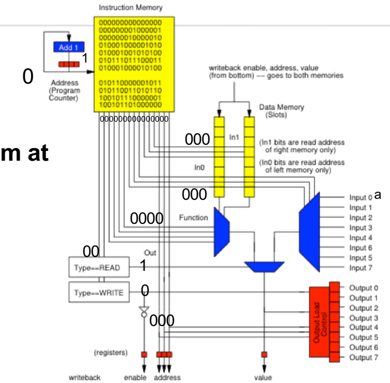


46

ESE150 Spring 2020

# Animate

- Start at PC=0
- Read Instr. Mem at 0
- Decode

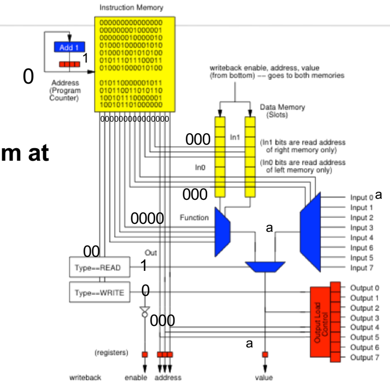


47

ESE150 Spring 2020

# Animate

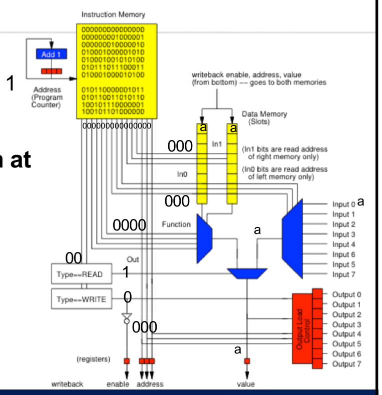
- Start at PC=0
- Read Instr. Mem at 0
- Decode
- From input



48

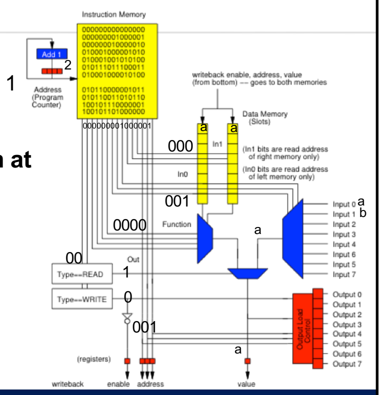
## Animate

- Start at PC=0
- Read Instr. Mem at 0
- Decode
- From input
- Write Back
- Update PC



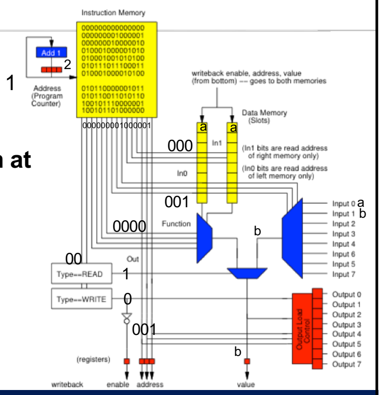
## Animate

- PC=1
- Read Instr. Mem at 1



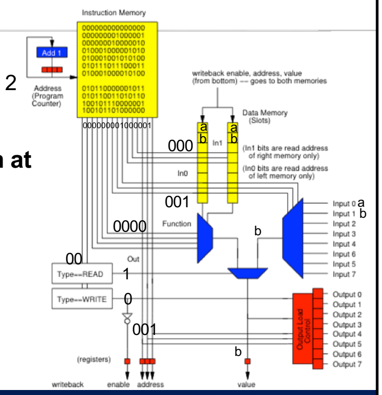
## Animate

- PC=1
- Read Instr. Mem at 1
- Decode
- From Input



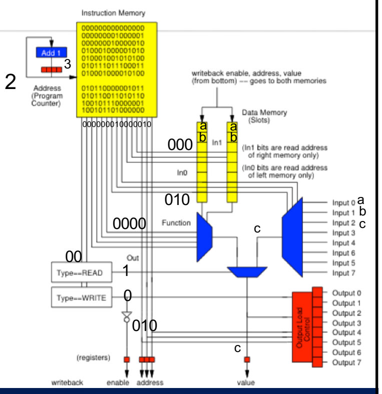
## Animate

- PC=1
- Read Instr. Mem at 1
- Decode
- From Input
- Writeback and update PC



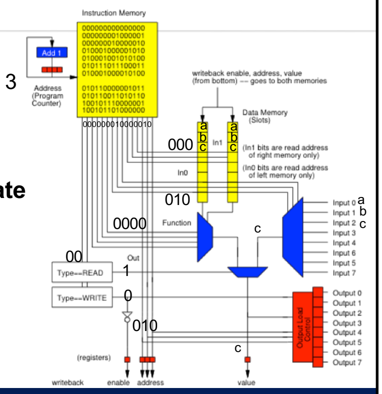
## Animate

- PC=2
- Another read



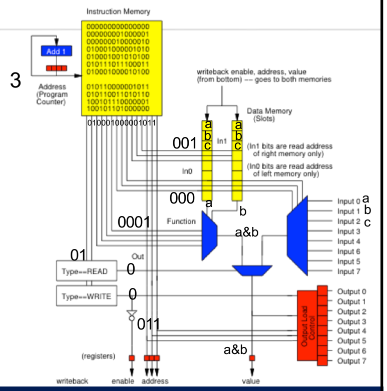
## Animate

- PC=2
- Another read
- Writeback, update PC



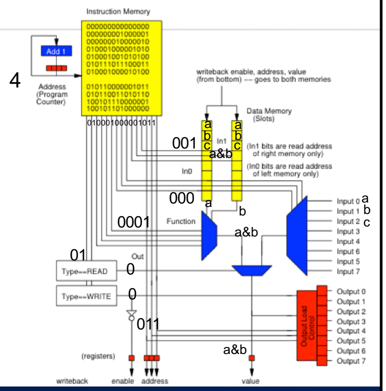
# Animate

- PC=3



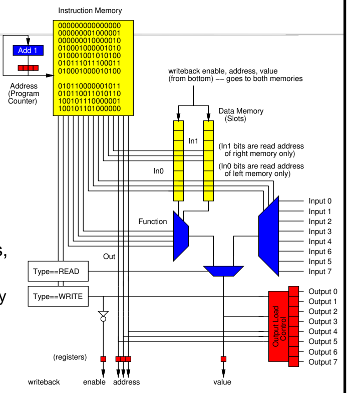
# Animate

- PC=3
- Writeback and update PC



# Processor

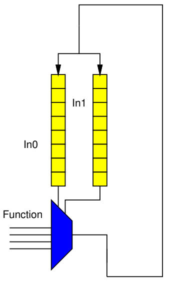
- Continue this sequence
  - Given PC
  - Read from Instruction Memory
  - Instruction bits control the datapath (memories, function, muxes)
    - Read from data memory
    - Perform operation
  - Write results back to memory; update PC



# Basic Idiom

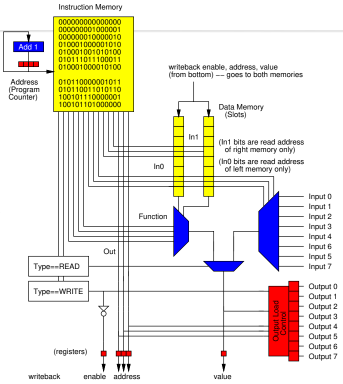
**Repeat:**

1. **Read gate value from memory**
2. **Perform operation on gate**
3. **Write result back to memory**



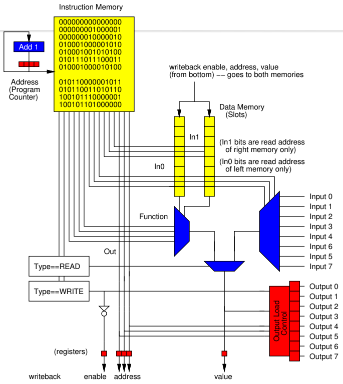
# Universal Processor

- **Can change computation simply be changing contents of instruction memory**



# Review

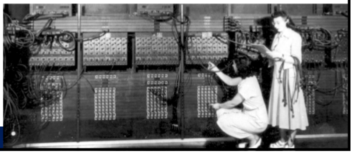
- **Single active compute element (programmable gate)**
- **Sequence in time**
- **Store state in memory**
- **Use Instruction memory to select and sequence operations**



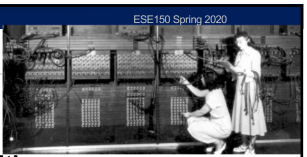
## Stored-Program Processor

## "Stored Program" Computer

- Can build physical machines that perform *any* computation.
- Can be built with limited hardware that is reused in time.
- Historically: this was a key contribution of Penn's Moore School
  - ENIAC→ EDVAC
  - Computer Engineers: Eckert and Mauchly
  - (often credited to Von Neumann)



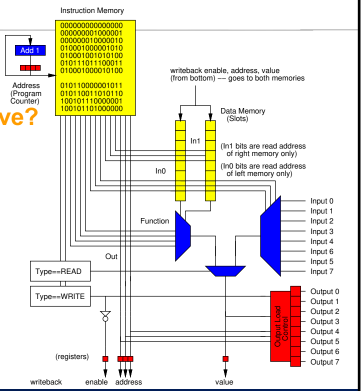
## Basic Idea



- Express computation in terms of a few primitives
  - E.g. Add, Multiply, OR, AND, NAND
- Provide one of each hardware primitive
- Store intermediates in memory
- Sequence operations on hardware to perform larger computation
- Store *description* of operation sequence in memory as well – hence "Stored Program"
- By filling in memory, can program to perform any computation

## Building Out

- How limited?
- How might improve?

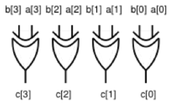


## Beyond Single Gate

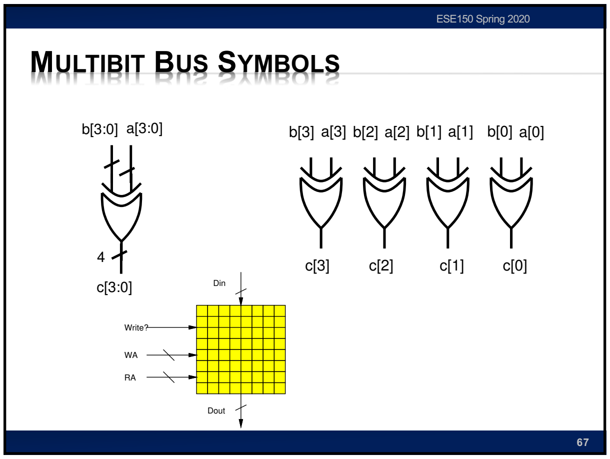
- Single gate extreme to make the high-level point
  - Except in some particular cases, not practical
- Usually reuse larger blocks
  - Adders
  - Multipliers
- Get more done per cycle than one gate
- Now it's a matter of engineering the design point
  - Where do we want to be between one gate and full circuit extreme?
  - How many gate evaluations should we physically compute each cycle?

## Word-Wide Processors

- Common to compute on multibit words
  - Add two 16b numbers
  - Multiply two 16b numbers
  - Perform bitwise-XOR on two 32b numbers
- More hardware
  - 16 full adders, 32 XOR gates



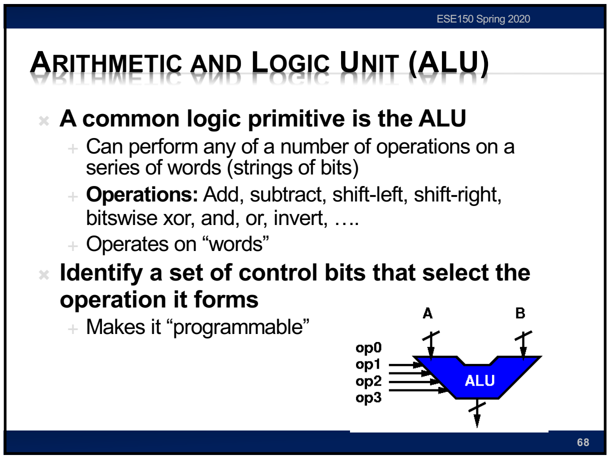
- All programmable gates doing the same thing
  - So don't require more instruction bits



ESE150 Spring 2020

## Multibit Bus Symbols

67



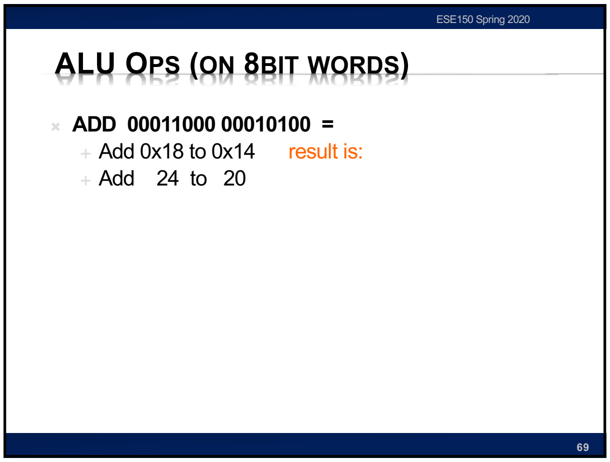
ESE150 Spring 2020

## Arithmetic and Logic Unit (ALU)

- **A common logic primitive is the ALU**
  - Can perform any of a number of operations on a series of words (strings of bits)
  - **Operations:** Add, subtract, shift-left, shift-right, bitswise xor, and, or, invert, ….
  - Operates on “words”
- **Identify a set of control bits that select the operation it forms**
  - Makes it “programmable”

A
B
op0
op1
op2
op3
ALU

68

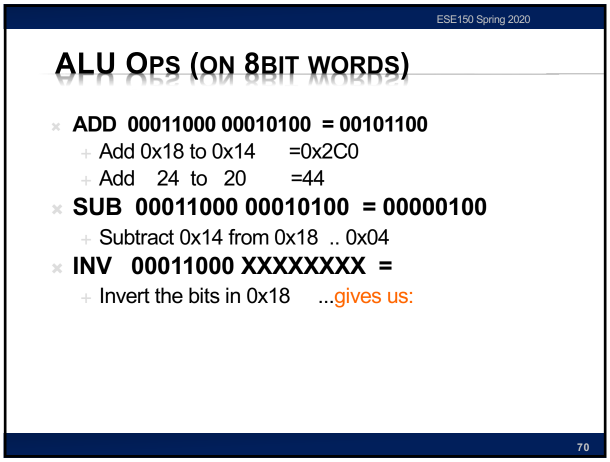


ESE150 Spring 2020

## ALU Ops (on 8bit words)

- **ADD 00011000 00010100 =**
  - Add 0x18 to 0x14 result is:
  - Add 24 to 20

69

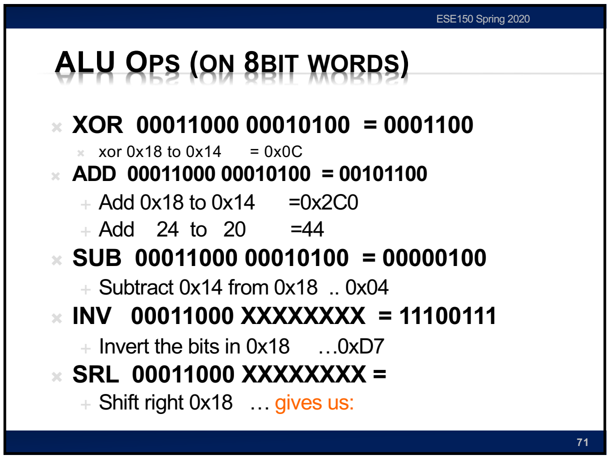


ESE150 Spring 2020

## ALU Ops (on 8bit words)

- **ADD 00011000 00010100 = 00101100**
  - Add 0x18 to 0x14 =0x2C0
  - Add 24 to 20 =44
- **SUB 00011000 00010100 = 00000100**
  - Subtract 0x14 from 0x18 .. 0x04
- **INV 00011000 XXXXXXXX =**
  - Invert the bits in 0x18 ...gives us:

70

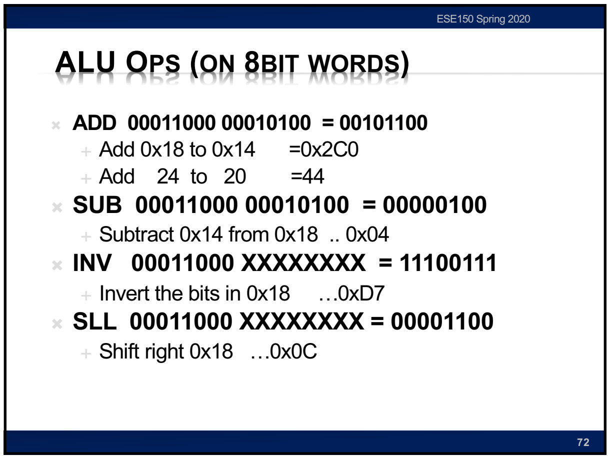


ESE150 Spring 2020

## ALU Ops (on 8bit words)

- **XOR 00011000 00010100 = 0001100**
  - xor 0x18 to 0x14 = 0x0C
- **ADD 00011000 00010100 = 00101100**
  - Add 0x18 to 0x14 =0x2C0
  - Add 24 to 20 =44
- **SUB 00011000 00010100 = 00000100**
  - Subtract 0x14 from 0x18 .. 0x04
- **INV 00011000 XXXXXXXX = 11100111**
  - Invert the bits in 0x18 …0xD7
- **SRL 00011000 XXXXXXXX =**
  - Shift right 0x18 … gives us:

71



ESE150 Spring 2020

## ALU Ops (on 8bit words)

- **ADD 00011000 00010100 = 00101100**
  - Add 0x18 to 0x14 =0x2C0
  - Add 24 to 20 =44
- **SUB 00011000 00010100 = 00000100**
  - Subtract 0x14 from 0x18 .. 0x04
- **INV 00011000 XXXXXXXX = 11100111**
  - Invert the bits in 0x18 …0xD7
- **SLL 00011000 XXXXXXXX = 00001100**
  - Shift right 0x18 …0x0C

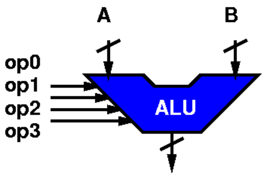
72

## ALU Ops (on 8bit words)

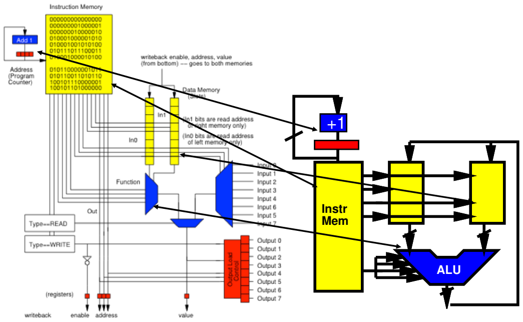
- ADD 00011000 00010100 = 00101100
  - Add 0x18 to 0x14 =0x2C0
  - Add 24 to 20 =44
- SUB 00011000 00010100 = 00000100
  - Subtract 0x14 from 0x18 .. 0x04
- INV 00011000 XXXXXXXX = 11100111
  - Invert the bits in 0x18 …0xD7
- SLL 00011000 XXXXXXXX = 00001100
  - Shift right 0x18 …0x0C
- XOR 00011000 00010100 = 0001100
  - xor 0x18 to 0x14 = 0x0C

## ALU Encoding

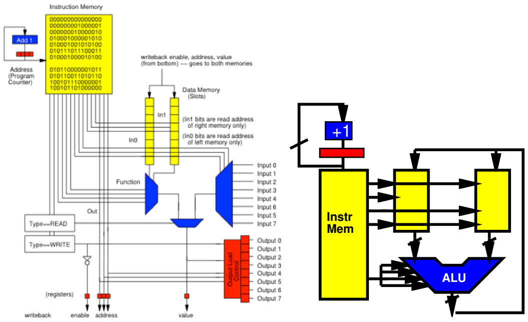
- Each operation has some bit sequence
- ADD 0000
- SUB 0010
- INV 0001
- SLL 1110
- SLR 1100
- AND 1000



## ALU-based Word-Wide Processor

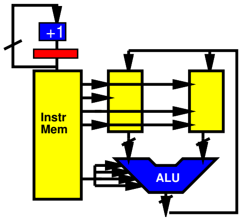


## ALU-based Word-Wide Processor



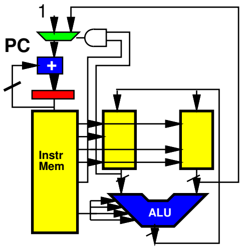
## Beyond Linear Sequence

- So far, processor can run a fixed sequence
- Cannot
  - Implement a loop
  - Implement an if-then-else



## Branching

- Allow PC to advance by value other than 1
  - Could be negative
- Allow data to impact selection
  - Only load when data bit is 1
- Add Instruction bits (or instruction) to control loading

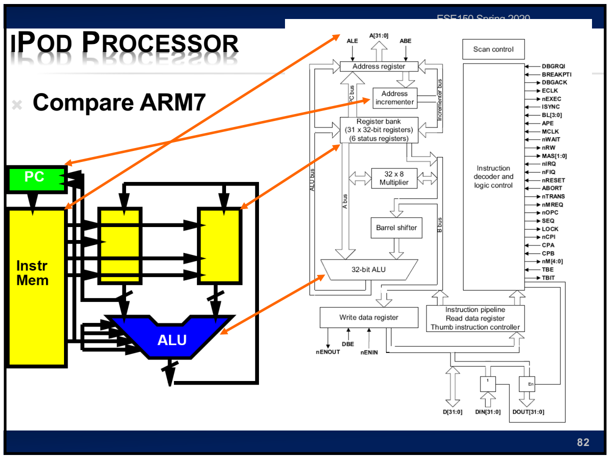


- BRANCH if (SRC1[0]==1) to PC+SRC2

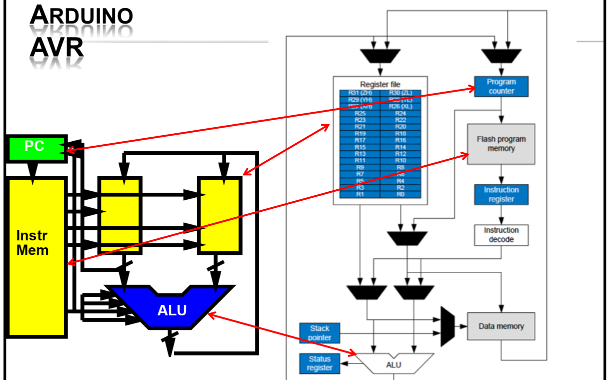
# Contemporary Processors

## iPod Processor

- Compare ARM7



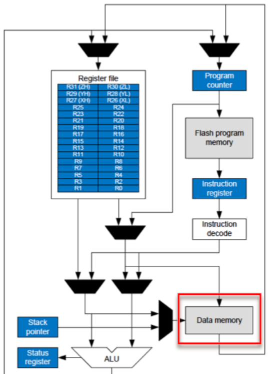
## Arduino AVR



ATmega328/P Datasheet

## Arduino AVR

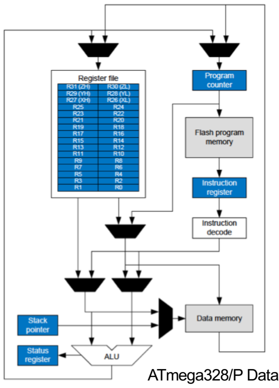
- Adds separate Data Memory from Register File
- (common, omitted above for simplicity)



ATmega328/P Datasheet

## Arduino AVR

- 8-bit architecture
  - 8b wide ALU
- 32x8 Register File
  - 32 register
  - 8b wide
- 16b instructions
  - “most” instructions
- 32KB program memory
  - Flash2KB data memory
  - SRAM

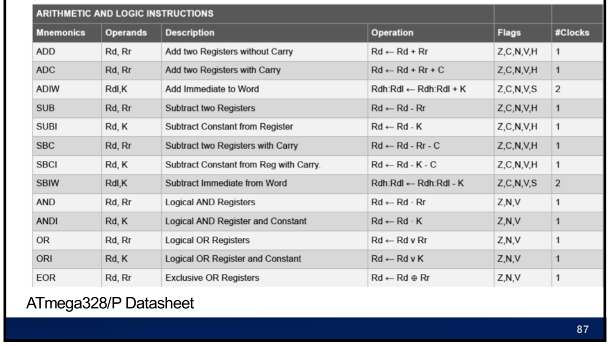


ATmega328/P Datasheet

## Instructions: Two Operand

- Arduino (AVR) has 2-operand, where one operand is both source and destination
- ADD R1, R2
  - Says: R1←R1+R2
- Use to make code more compact

# AVR Instructions



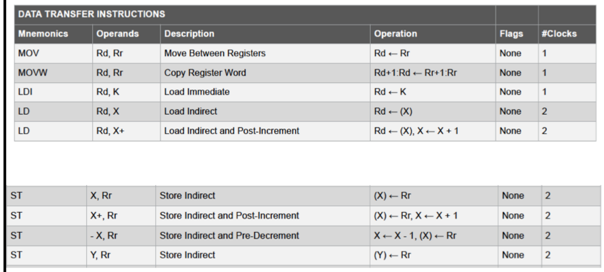
| ARITHMETIC AND LOGIC INSTRUCTIONS | | | | | |
|---|---|---|---|---|---|
| **Mnemonics** | **Operands** | **Description** | **Operation** | **Flags** | **#Clocks** |
| ADD | Rd, Rr | Add two Registers without Carry | $Rd \leftarrow Rd + Rr$ | Z,C,N,V,H | 1 |
| ADC | Rd, Rr | Add two Registers with Carry | $Rd \leftarrow Rd + Rr + C$ | Z,C,N,V,H | 1 |
| ADIW | Rdl,K | Add Immediate to Word | $Rdh:Rdl \leftarrow Rdh:Rdl + K$ | Z,C,N,V,S | 2 |
| SUB | Rd, Rr | Subtract two Registers | $Rd \leftarrow Rd - Rr$ | Z,C,N,V,H | 1 |
| SUBI | Rd, K | Subtract Constant from Register | $Rd \leftarrow Rd - K$ | Z,C,N,V,H | 1 |
| SBC | Rd, Rr | Subtract two Registers with Carry | $Rd \leftarrow Rd - Rr - C$ | Z,C,N,V,H | 1 |
| SBCI | Rd, K | Subtract Constant from Reg with Carry. | $Rd \leftarrow Rd - K - C$ | Z,C,N,V,H | 1 |
| SBIW | Rdl,K | Subtract Immediate from Word | $Rdh:Rdl \leftarrow Rdh:Rdl - K$ | Z,C,N,V,S | 2 |
| AND | Rd, Rr | Logical AND Registers | $Rd \leftarrow Rd \cdot Rr$ | Z,N,V | 1 |
| ANDI | Rd, K | Logical AND Register and Constant | $Rd \leftarrow Rd \cdot K$ | Z,N,V | 1 |
| OR | Rd, Rr | Logical OR Registers | $Rd \leftarrow Rd \vee Rr$ | Z,N,V | 1 |
| ORI | Rd, K | Logical OR Register and Constant | $Rd \leftarrow Rd \vee K$ | Z,N,V | 1 |
| EOR | Rd, Rr | Exclusive OR Registers | $Rd \leftarrow Rd \oplus Rr$ | Z,N,V | 1 |

ATmega328/P Datasheet

# AVR Instructions (Lab Appendix)

| Instruction Encoding | Instruction Description | Min Cycles | Max Cycles |
|---|---|---|---|
| add Rd, Rr | Add without carry: Rd = Rr + Rd and C is set to the carry-out bit | 1 | 1 |
| adc Rd, Rr | Add with carrying: Rd = Rd + Rr + C (which was previously set); C is set to the new carry-out bit | 1 | 1 |
| and Rd, Rr | Logical And: Rd = Rd AND Rr | 1 | 1 |
| brne OFFSET | Branch Not Equal: If Z=0, Move the instruction execution (back or forward) by OFFSET. | 1 | 2 |
| brpl OFFSET | Branch if Positive: If N=0, Move the instruction execution (back or forward) by OFFSET. | 1 | 2 |

# Data Memory Read / Write (Load/Store)



| DATA TRANSFER INSTRUCTIONS | | | | | |
|---|---|---|---|---|---|
| **Mnemonics** | **Operands** | **Description** | **Operation** | **Flags** | **#Clocks** |
| MOV | Rd, Rr | Move Between Registers | $Rd \leftarrow Rr$ | None | 1 |
| MOVW | Rd, Rr | Copy Register Word | $Rd+1:Rd \leftarrow Rr+1:Rr$ | None | 1 |
| LDI | Rd, K | Load Immediate | $Rd \leftarrow K$ | None | 1 |
| LD | Rd, X | Load Indirect | $Rd \leftarrow (X)$ | None | 2 |
| LD | Rd, X+ | Load Indirect and Post-Increment | $Rd \leftarrow (X), X \leftarrow X + 1$ | None | 2 |
| ST | X, Rr | Store Indirect | $(X) \leftarrow Rr$ | None | 2 |
| ST | X+, Rr | Store Indirect and Post-Increment | $(X) \leftarrow Rr, X \leftarrow X + 1$ | None | 2 |
| ST | - X, Rr | Store Indirect and Pre-Decrement | $X \leftarrow X - 1, (X) \leftarrow Rr$ | None | 2 |
| ST | Y, Rr | Store Indirect | $(Y) \leftarrow Rr$ | None | 2 |

ATmega328/P Datasheet

# Next Lab

- **Look at Instruction-Level code for Arduino**
- **Understand performance from instruction-level code**
- **Need to download Arduino IDE for your computer**

# Big Ideas

- **Memory stores data compactly**
- **Can implement large computations on small hardware by reusing hardware in time**
  - Storing computational state in memory
- **Can store program control in instruction memory**
  - Change program by reprogramming memory
  - Universal machine: Stored-Program Processor

# Learn More

- **CIS240 – processor organization and assembly**
- **CIS371 – implement and optimize processors**
  - Including FPGA mapping in Verilog
- **ESE370 – implement memories (and gates) using transistors**