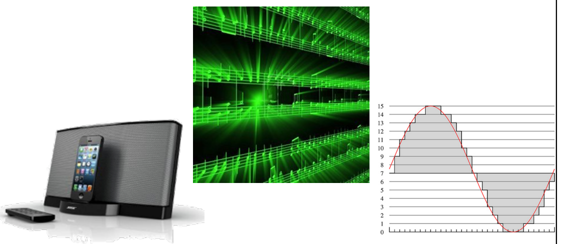


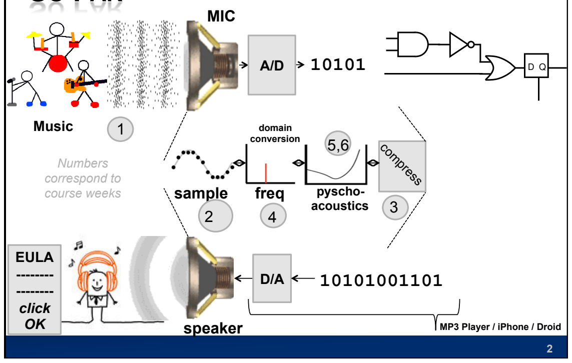
Lecture #8 – Stored-Program Processors

ESE 150 – Digital Audio Basics

ESE150 Spring 2018

Based on slides © 2009--2018 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.

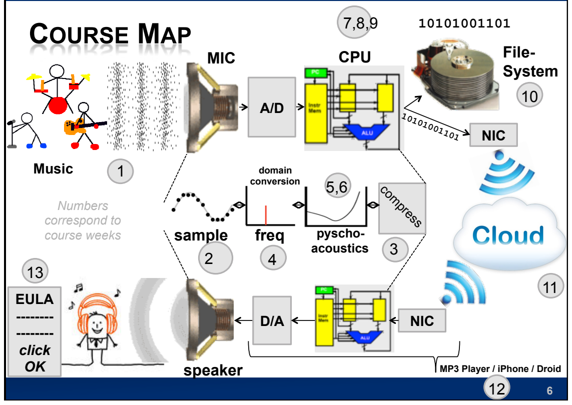
Economy and Universality

- **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?**

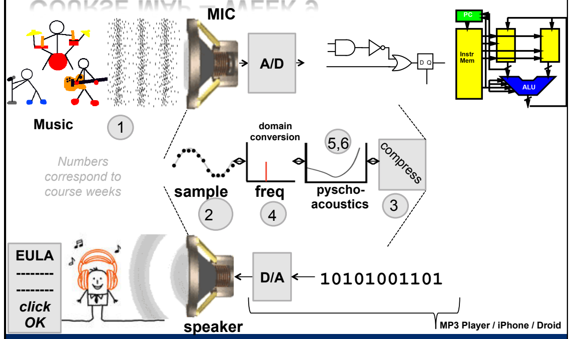
Lecture Topics

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

Course Map



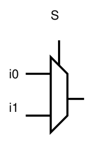
Course Map – Week 9



Quick Reminder

Multiplexer Gate

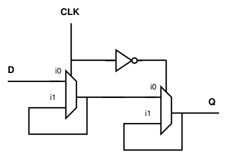
- **MUX**
  - When S=0, output=i0
  - When S=1, output=i1

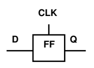


S	i0	i1	Mux2(S,i0,i1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

State Element

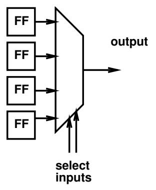
- **Latch or Register is a state element**
- **Allows circuit to *remember* a value**
- **Build computations that**
  - Depend on past inputs
  - Reuse hardware in time





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)

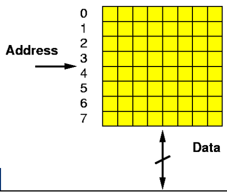
Preclass 1

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

Memory

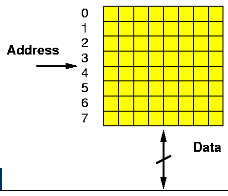
Random Access Memory

- A Memory:
  - Series of locations
  - Can write values into
  - Read values from
  - Return last value written

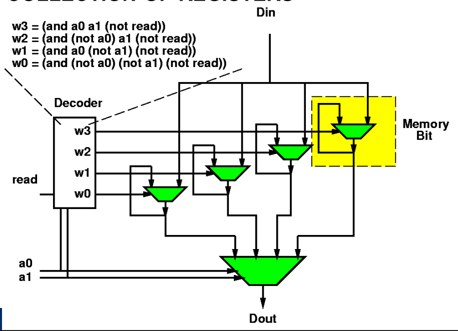


Two Pieces of a Memory

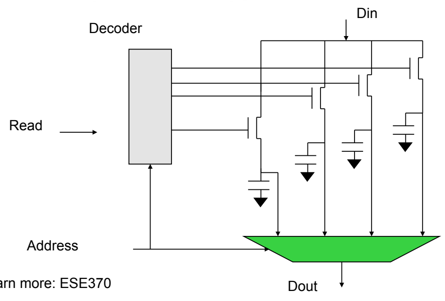
1. Element to remember a value
2. Way to address/select that element



Could build Memory w/ Muxes & latches … collection of registers


Din
w3 = (and a0 a1 (not read))
w2 = (and (not a0) a1 (not read))
w1 = (and a0 (not a1) (not read))
w0 = (and (not a0) (not a1) (not read))
Decoder
w3
w2
w1
w0
read
Memory Bit
a0
a1
Dout

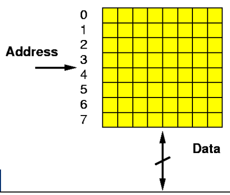
Random Access Memory (RAM) with Capacitor Memories



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



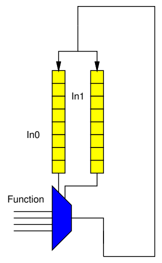
ONE-GATE PROCESSOR

IDEA

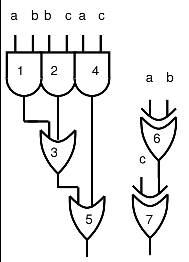
- Store register and gate outputs in memory
- Compute one gate at a time
  - Using a single physical gate

BASIC IDIOM

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

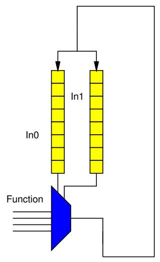


OPERATION



```
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	

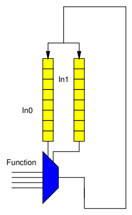


OPERATION SEQUENCE

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

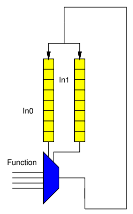
C	Description	Instruction Fields Type	Function	In0	In1	Out
a=getInput(0);	read input 0 and put in slot 0	READ	NONE	0	0	0
b=getInput(1);	read input 1 and put in slot 1	READ	NONE	1	0	1
c=getInput(2);	read input 2 and put in slot 2	READ	NONE	2	0	2
`t1=a&b;`	read value in slot 0 and value in slot 1, perform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

Missing description?



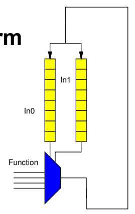
Observe

- 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?)



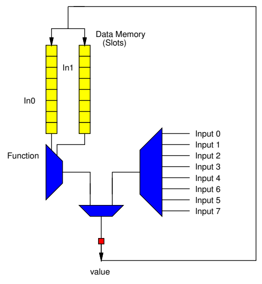
Instruction

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



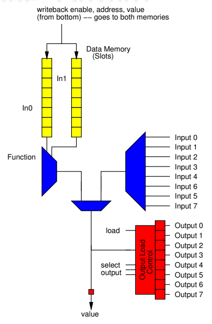
Expanding the Structure: Input

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



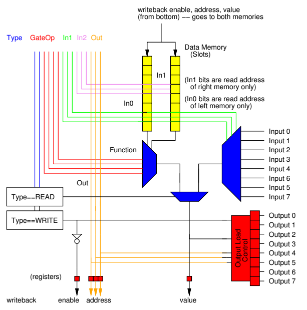
Expanding the Structure: Output

- Add way to load a designated output register



Expanded Control = Instruction

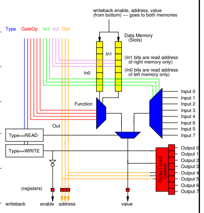
- Group the full control into instruction
- Set of bits that tells the structure what to do



Fillin missing Instruction

C	Description	Instruction Fields 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	2
	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 2 and value in slot 3, 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 2 and value in slot 3, 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					

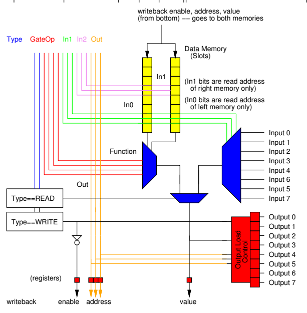
0	1	2	3	4	5	6	7
a	b	c	t1	t2	o1	o2	


writeback enable, address, value
(from bottom) –– goes to both memories
Type
GateOp
In1
In2
Out
Data Memory
(Slots)
In1
(In1 bits are read address of right memory only)
(In0 bits are read address of left memory only)
In0
Function
Type==READ
Type==WRITE
(registers)
writeback
enable
address
value

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


writeback enable, address, value (from bottom) -- goes to both memories
Type GateOp In1 In2 Out
Data Memory (Slots)
In1
(In1 bits are read address of right memory only)
In0
(In0 bits are read address of left memory only)
Function
Out
Input 0 … Input 7
Type==READ
Type==WRITE
Output 0 … Output 7
Output Load Control
(registers)
writeback enable address value

Instruction Bits Example

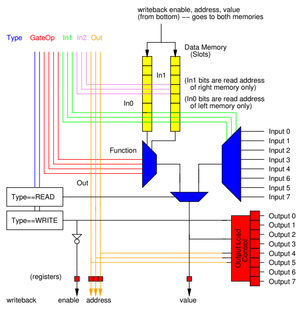
- Fillin Missing

GATE	AND	0	1	2	010001000001010

`t1=t1|t2;`	read value in slot 2 and value in slot 3, 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 2 and value in slot 3, perform an OR on the values, and store into slot 5	GATE	OR	3	4	5	

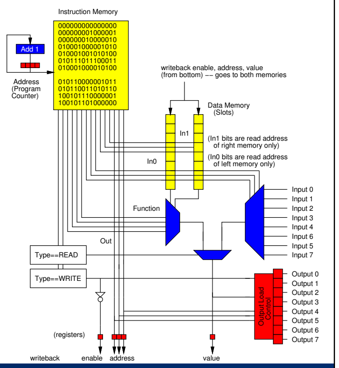
Instruction Sequence Control

- How provide the sequence of instructions?



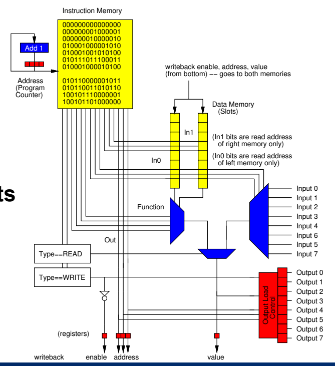
Instruction Memory

- Add Memory to hold set of Instructions
- Counter to sequence instructions


Instruction Memory
Add 1
Address (Program Counter)
writeback enable, address, value (from bottom) -- goes to both memories
Data Memory (Slots)
In1
In0
Function
Out
Type==READ
Type==WRITE
Output Load Control
(registers)
writeback enable address value

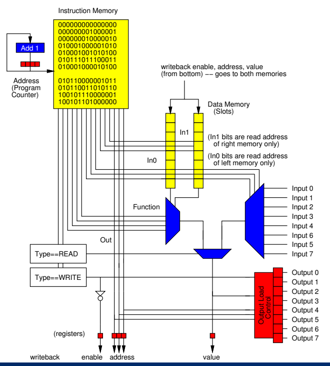
Universal Processor

- Can change computation simply be changing contents of instruction memory



Review

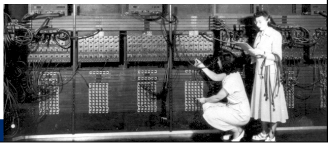
- Single active compute element (programmable gate)
- Sequence in time
- Store state in memory
- Use Instruction memory to select and sequence operations



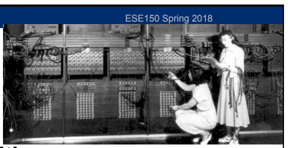
Stored-Program Processor

"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)



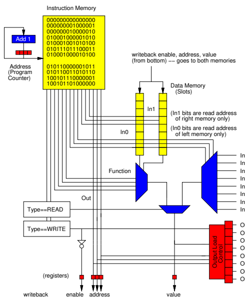
Basic Idea



- 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?

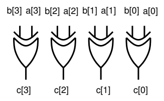


Beyond Single Gate

- 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?

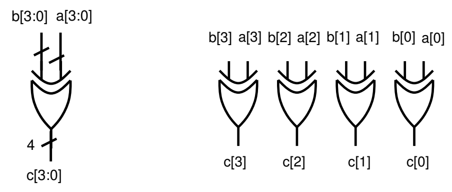
Word-Wide Processors

- 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



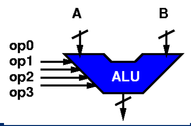
- All programmable gates doing the same thing
  - So don't require more instruction bits

Multibit Bus Symbols



Arithmetic and Logic Unit (ALU)

- **A common primitive logic 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”

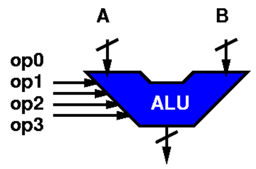


ALU Ops (on 8bit words)

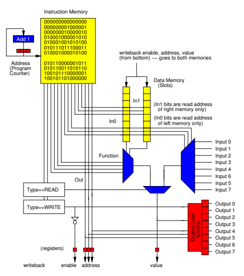
- **XOR 00011000 00010100 =**
  - xor 0x18 to 0x14 result is:
- **ADD 00011000 00010100 =**
  - Add 0x18 to 0x14 result is:
  - Add 24 to 20 result is:
- **SUB 00011000 00010100 =**
  - Subtract 0x14 from 0x18 …result is:
- **INV 00011000 XXXXXXXX =**
  - Invert the bits in 0x18 ...gives us:
- **SLL 00011000 XXXXXXXX =**
  - Shift left 0x18 … gives us:

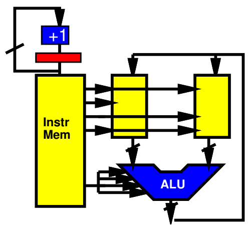
ALU Encoding

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



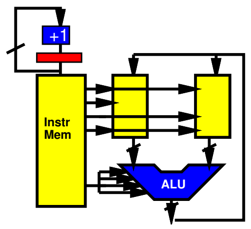
ALU-based Word-Wide Processor





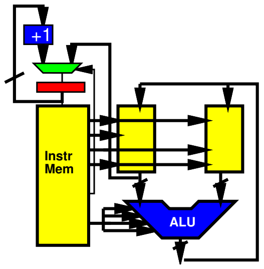
Beyond Linear Sequence

- **So far, processor can run a fixed sequence**
- **Might like to**
  - Repeat sequence
  - Conditionally execute instruction or sequence



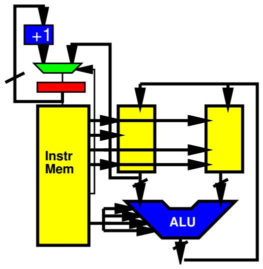
Branching

- Allow PC to be loaded
- Add Instruction bits (or instruction) to control loading
- BRANCH Slot=In0



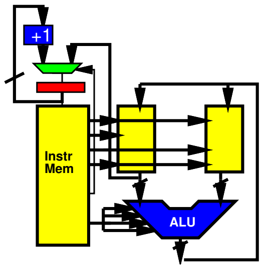
Branching

- How
  - Branch to top of loop?
  - Conditionally branch to top of loop?
  - Implement if-then?



Branching

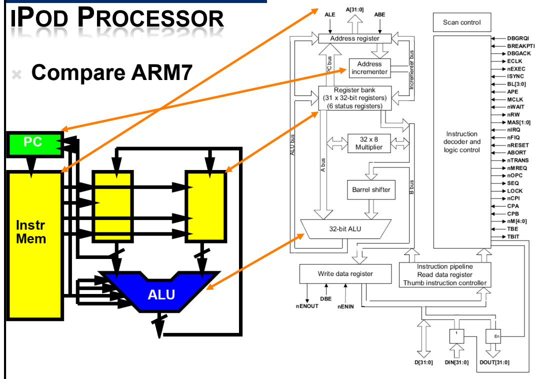
- Conditional in slot 4
- Slot 7 – true target
- Slot 8 – false target
- S5=S4<<1
- S4=S5|S4
- *(repeat width)*
- S5=!S4
- S6=S4*S7 // 0 or S7
- S5=S5*S8 // S8 or 0
- S5=S5+S6 // target
- BRANCH S5



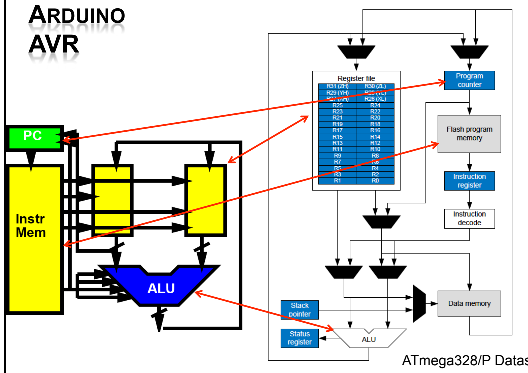
Contemporary Processors

iPod Processor

- Compare ARM7



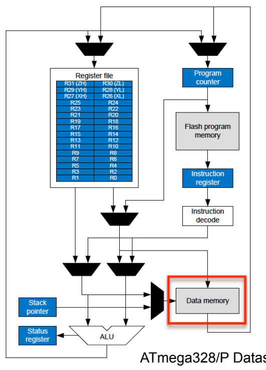
Arduino AVR



ATmega328/P Datasheet

Arduino AVR

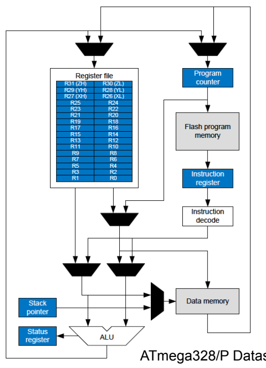
- Adds separate Data Memory from Register File
- (common, omitted above for simplicity)



ATmega328/P Datasheet

Arduino AVR

- 8-bit architecture
  - 8b wide ALU
- 32x8 Register File
  - 32 register
  - 8b wide
- 16b instructions
  - "most" instructions
- 2K B data memory
  - SRAM
- 32KB program memory
  - Flash



ATmega328/P Datasheet

Instructions: Two Operand

- Typically 2-operand, where one operation 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 ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add two Registers with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract two Registers with Carry	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract Constant from Reg with Carry.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd · Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd · K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1

ATmega328/P Datasheet

Branching Instructions

BRANCH INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clocks
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP(1)	k	Direct Jump	PC ← k	None	3

BRANCH INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clocks
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL(1)	k	Direct Subroutine Call	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4

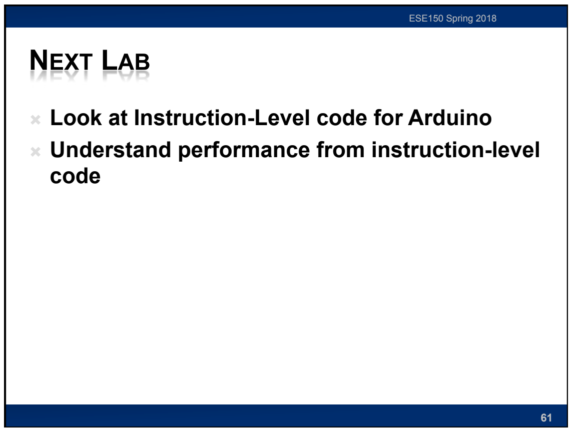
ATmega328/P Datasheet

Data Memory Read / Write (Load/Store)

DATA TRANSFER INSTRUCTIONS					
Mnemonics	Operands	Description	Operation	Flags	#Clocks
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Increment	Rd ← (X), X ← X + 1	None	2

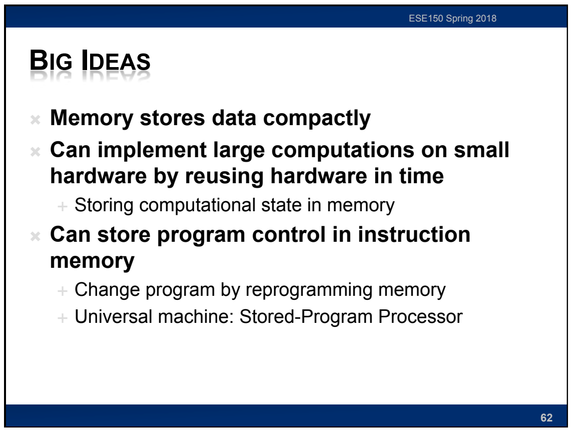
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Increment	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store Indirect and Pre-Decrement	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2

ATmega328/P Datasheet



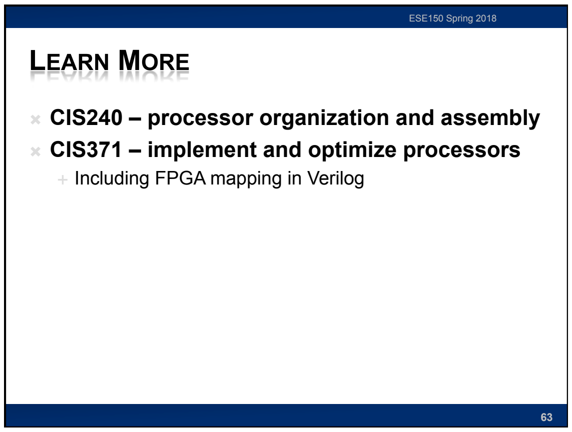
Next Lab

- Look at Instruction-Level code for Arduino
- Understand performance from instruction-level code



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