

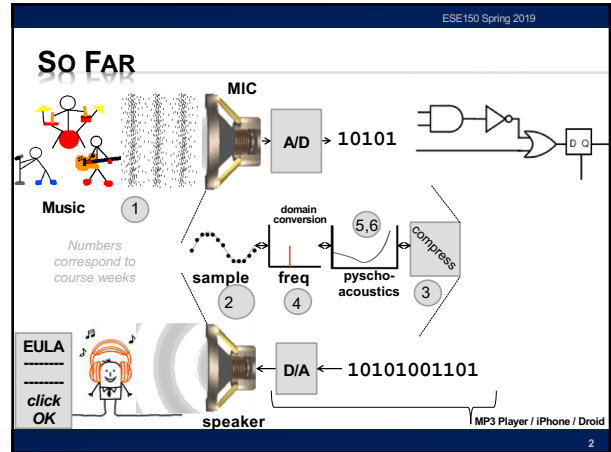
Penn Engineering **ESE**

Lecture #8 – Stored-Program Processors

**ESE 150 – DIGITAL AUDIO BASICS**

ESE150 Spring 2019

Based on slides © 2009–2019 DeHon



ESE150 Spring 2019

### 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.

3

ESE150 Spring 2019

### 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?

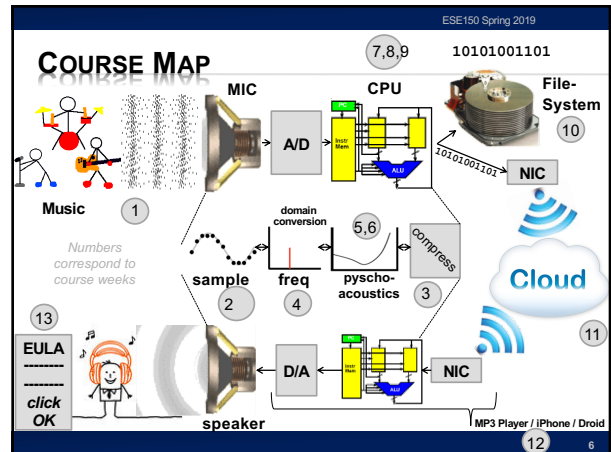
4

ESE150 Spring 2019

### LECTURE TOPICS

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

5



ESE150 Spring 2019

## COURSE MAP – WEEK 9

Music 1  
Numbers correspond to course weeks

sample 2  
freq 4  
psycho-acoustics 3  
compress

EULA  
click OK

speaker

MP3 Player / iPhone / Droid

7

ESE150 Spring 2019

## QUICK REMINDER

8

ESE150 Spring 2019

## MULTIPLEXER GATE

- × **MUX**
  - + When  $S=0$ , output= $i_0$
  - + When  $S=1$ , output= $i_1$

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

9

ESE150 Spring 2019

## 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

10

ESE150 Spring 2019

## 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	

11

ESE150 Spring 2019

## 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)

12

ESE150 Spring 2019

## PRECLASS 1

- × **What Function?**
  - +  $o1 = a \& b \mid b \& c \mid a \& c;$
  - +  $o2 = a \wedge b \wedge c;$
- × **How many gates?**

13

ESE150 Spring 2019

## PRECLASS 1 IN GATES

14

ESE150 Spring 2019

## MEMORY

15

ESE150 Spring 2019

## RANDOM ACCESS MEMORY

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

Notation:  
 slash on wire  
 means multiple bits wide

16

ESE150 Spring 2019

## TWO PIECES OF A MEMORY

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

17

ESE150 Spring 2019

## COULD BUILD MEMORY W/ MUXES & LATCHES

### ... COLLECTION OF REGISTERS

$w3 = (\text{and } a0 \ a1 \ (\text{not read}))$   
 $w2 = (\text{and } (\text{not } a0) \ a1 \ (\text{not read}))$   
 $w1 = (\text{and } a0 \ (\text{not } a1) \ (\text{not read}))$   
 $w0 = (\text{and } (\text{not } a0) \ (\text{not } a1) \ (\text{not read}))$

18

ESE150 Spring 2019

## RANDOM ACCESS MEMORY (RAM) WITH CAPACITOR MEMORIES

Learn more: ESE370

19

ESE150 Spring 2019

## KEY ENGINEERING PROPERTY

- × **Store state compactly in memory**
- × **A(memory cell) small**
  - +  $A(\text{mem}) < A(\text{gate})$
- × **Depends on few inputs/outputs**
  - + Memory cells share inputs and outputs

22

ESE150 Spring 2019

## ONE-GATE PROCESSOR

21

ESE150 Spring 2019

## IDEA

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

22

ESE150 Spring 2019

## BASIC IDIOM

**Repeat:**

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

23

ESE150 Spring 2019

## OPERATION

a b b c a c

```

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	

24

ESE150 Spring 2019

## 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	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;	Missing description?	GATE	AND	0	2	4	

25

ESE150 Spring 2019

## 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?)

26

ESE150 Spring 2019

## INSTRUCTION

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

27

ESE150 Spring 2019

## EXPANDING THE STRUCTURE: INPUT

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

28

ESE150 Spring 2019

## EXPANDING THE STRUCTURE: OUTPUT

- ✗ Add way to load a designated output register

29

ESE150 Spring 2019

## EXPANDED CONTROL = INSTRUCTION

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

30

### FILLIN MISSING INSTRUCTION

C	Description	Instruction Fields	In	Out	
Type	Function	GateOp	In1	In2	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=a&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 1 and value in slot 2, perform an AND on the values, and store into slot 5	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

### INSTRUCTION BITS

GATE AND 0 1 2 01000100001010

- 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 01000100001010

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	01011011100011
t2=a&c;	read value in slot 1 and value in slot 2, perform an AND on the values, and store into slot 5	GATE AND	0	2	4	01000100001010
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 SEQUENCE CONTROL

How provide the sequence of instructions?

### INSTRUCTION MEMORY

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

### ANIMATE

Start at PC=0

ESE150 Spring 2019

# ANIMATE

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

37

ESE150 Spring 2019

# ANIMATE

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

38

ESE150 Spring 2019

# ANIMATE

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

39

ESE150 Spring 2019

# ANIMATE

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

40

ESE150 Spring 2019

# ANIMATE

- PC=1
- Read Instr. Mem at 1

41

ESE150 Spring 2019

# ANIMATE

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

42

### ANIMATE

ESE150 Spring 2019

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

43

### ANIMATE

ESE150 Spring 2019

- PC=2
- Another read

44

### ANIMATE

ESE150 Spring 2019

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

45

### ANIMATE

ESE150 Spring 2019

- PC=3

46

### ANIMATE

ESE150 Spring 2019

- PC=3
- Writeback and update PC

47

### PROCESSOR

ESE150 Spring 2019

- 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

48



ESE150 Spring 2019

## BASIC IDIOM

Repeat:

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

49

ESE150 Spring 2019

## UNIVERSAL PROCESSOR

Can change computation simply by changing contents of instruction memory

50

ESE150 Spring 2019

## REVIEW

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

51

ESE150 Spring 2019

## STORED-PROGRAM PROCESSOR

52

ESE150 Spring 2019

## “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)

53

ESE150 Spring 2019

## 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

54

ESE150 Spring 2019

## BUILDING OUT

- How limited?
- How might improve?

55

ESE150 Spring 2019

## 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?

56

ESE150 Spring 2019

## 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

57

ESE150 Spring 2019

## MULTIBIT BUS SYMBOLS

58

ESE150 Spring 2019

## 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, bitwise xor, and, or, invert, ....
  - Operates on "words"
- Identify a set of control bits that select the operation it forms
  - Makes it "programmable"

59

ESE150 Spring 2019

## ALU OPS (ON 8BIT WORDS)

- XOR 00011000 00010100 =
  - xor 0x18 to 0x14 result is:

60

ESE150 Spring 2019

### ALU OPS (ON 8BIT WORDS)

- \* **XOR 00011000 00010100 = 0001100**
  - \* xor 0x18 to 0x14 = 0x0C
- \* **ADD 00011000 00010100 =**
  - + Add 0x18 to 0x14 **result is:**
  - + Add 24 to 20

61

ESE150 Spring 2019

### 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 =**
  - + Invert the bits in 0x18 ...gives us:

62

ESE150 Spring 2019

### 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
- \* **SLL 00011000 XXXXXXXX =**
  - + Shift left 0x18 ... gives us:

63

ESE150 Spring 2019

### 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
- \* **SLL 00011000 XXXXXXXX = 00110000**
  - + Shift left 0x18 ...0x30

64

ESE150 Spring 2019

### ALU ENCODING

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

op0

op1

op2

op3

65

ESE150 Spring 2019

### ALU-BASED WORD-WIDE PROCESSOR

66

ESE150 Spring 2019

## BEYOND LINEAR SEQUENCE

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

67

ESE150 Spring 2019

## 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

68

ESE150 Spring 2019

## BRANCHING

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

69

ESE150 Spring 2019

## CONTEMPORARY PROCESSORS

70

ESE150 Spring 2019

## IPOD PROCESSOR

- × Compare ARM7

71

ESE150 Spring 2019

## ARDUINO AVR

ATmega328/P Datasheet

72

ESE150 Spring 2019

## ARDUINO AVR

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

ATmega328/P Datasheet

73

ESE150 Spring 2019

## ARDUINO AVR

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

ATmega328/P Datasheet

74

ESE150 Spring 2019

## INSTRUCTIONS: TWO OPERAND

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

75

ESE150 Spring 2019

## 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
ADW	Rd, K	Add Immediate to Word	$Rd \leftarrow Rd + K$	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
SBW	Rd, K	Subtract Immediate from Word	$Rd \leftarrow Rd - K$	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	$Rd \leftarrow Rd \wedge Rr$	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	$Rd \leftarrow Rd \wedge K$	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

76

ESE150 Spring 2019

## AVR INSTRUCTIONS (LAB APPENDIX)

Instruction Encoding	Instruction Description	Number of Cycles to execute
add Rd, Rr	Add without carry: $Rd = Rr + Rd$ and C is set to the carry-out bit	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
and Rd, Rr	Logical And: $Rd = Rd \wedge Rr$	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

77

ESE150 Spring 2019

## 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
MOW	Rd, Rr	Copy Register Word	$Rd + 1 \leftarrow Rr + 1$	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

78

## NEXT LAB

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

79

## 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

80

## LEARN MORE

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

81