Chapter 5
(and some of Ch. 4)
The Von Neumann Model & LC3

Based on slides © McGraw-Hill
Additional material © 2004/2005/2006 Lewis/Martin

What Do We Know?

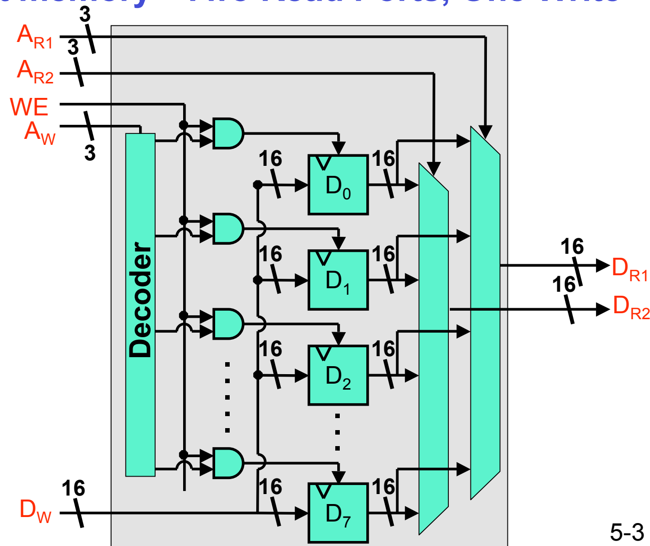
A LOT!!

- **Data representation** (binary, 2's complement, floating point, …)
- **Transistors** (p-type, n-type, CMOS)
- **Gates** (complementary logic)
- **Combinational logic circuits** (PLAs), **memory** (latches, flip-flops, …)
- **Sequential logic circuits** (state machines)
- **Simple "processors"** (programmable traffic sign)

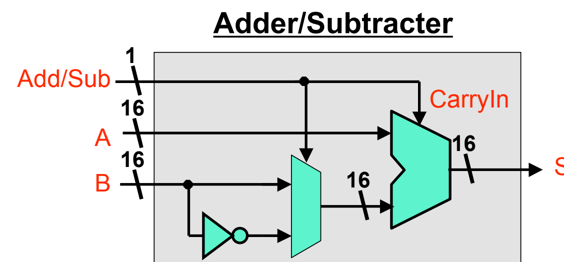
What's next?

- **Apply all this to traditional computing**
- **Software interface: instructions**
- **Hardware implementation: data path**

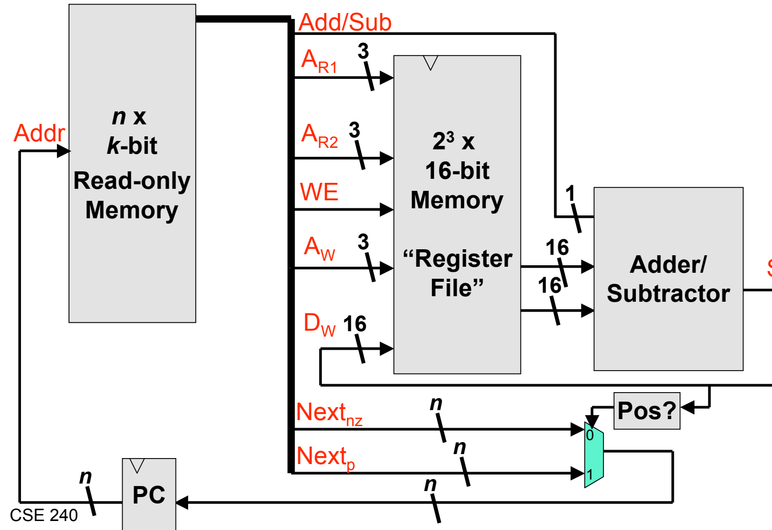
2^3 by 16-bit memory - Two Read Ports, One Write



16-bit Adder/Subtracter



Simple Processing Machine



Can we make it multiply?

Goal: A * B into C

While (B > 0)
  C = C + A
  B = B - 1

Initial register values

- **R2 is "A", R3 is "B", R4 will be "C"**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	$Next_{nz}$	$Next_p$
#0							
#1							
#2							
#3							

Can we make it multiply?

Goal: A * B into C

While (B > 0)
  C = C + A
  B = B - 1

Initial register values

- **R2 is "A", R3 is "B", R4 will be "C"**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	$Next_{nz}$	$Next_p$
#0	0	R3	R0	0	X	#3	#1
#1	0	R4	R2	1	R4	#2	#2
#2	1	R3	R1	1	R3	#0	#0
#3	X	X	X	0	X	#3	#3

Multiply Execution Trace

Cycle	0	1	2	3	4	5	6	7
PC	#0	#1	#2	#0	#1	#2	#0	#3
R0	0							
R1	1							
R2 ("A")	5							
R3 ("B")	2			1			0	
R4 ("C")	0		5			10		
R5	---							
R6	---							
R7	---							

Can we make it divide?

Goal: A / B into C

Initial register values

- **R2 is “A”, R3 is “B”, R4 will be “C”**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	$Next_{nz}$	$Next_p$
0							
1							
2							
3							

Can we make it divide?

A = A - B
If (A+1 > 0) ← A >= 0
C = C + 1
goto start

Goal: A / B into C

Initial register values

- **R2 is “A”, R3 is “B”, R4 will be “C”**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	$Next_{nz}$	$Next_p$
0	1	R2	R3	1	R2	#1	#1
1	0	R2	R1	0	X	#3	#2
2	0	R4	R1	1	R4	#0	#0
3	X	X	X	0	X	#3	#3

Divide Execution Trace

Cycle	0	1	2	3	4	5	6	7	8
PC	#0	#1	#2	#0	#1	#2	#0	#1	#3
R0	0								
R1	1								
R2 (“A”)	10	5			0			-5	
R3 (“B”)	5								
R4 (“C”)	0			1			2		
R5	---								
R6	---								
R7	---								

How might we improve our processing machine?

More operations & conditions

- **And, Not?**
- **Control next operation via any of negative/positive/zero combinations**

More data storage?

- **Add a separate data memory structure**
- **Register file used as “temporary” storage**
- **Add new logic elements to read and write this memory**

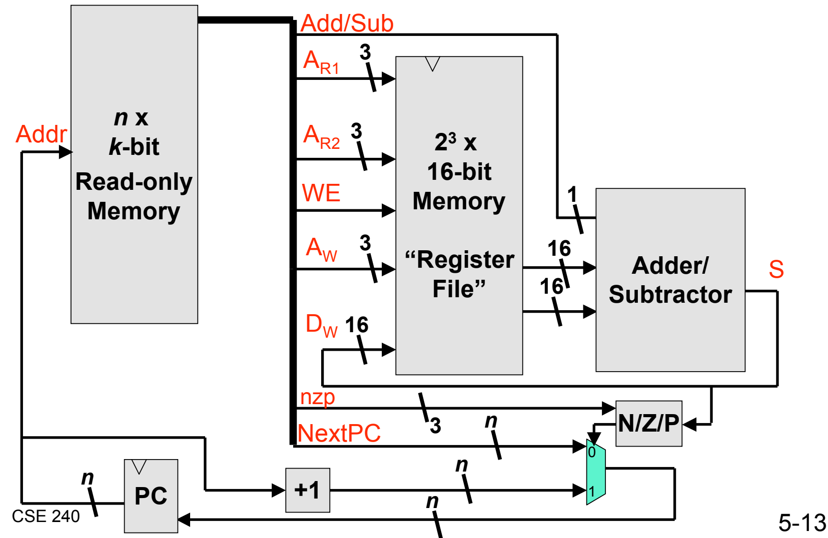
How would we sum all the numbers in memory?

- **Need “addressing modes” to allow this**
- **E.g., Read from the location in memory specified by R1**

Smaller encoding

- **Use fewer bits for “Next” (too large when control memory is big)**
- **Also want more “dynamic” control**
  - **E.g., next operation is at the location specified by R1’s value**

Simple Processing Machine -- Modified



Multiply for Modified Machine

Goal: A * B into C

Initial register values

- **R2 is “A”, R3 is “B”, R4 will be “C”**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

While (B > 0)
  C = C + A
  B = B - 1

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	N/Z/P	Next
#0	0	R3	R0	0	X	**110**	**#3**
#1	0	R4	R2	1	R4	**000**	**X**
#2	1	R3	R1	1	R3	**111**	**#0**
#3	X	X	X	0	X	**111**	**#3**

Divide for Modified Machine

Goal: A / B into C

Initial register values

- **R2 is “A”, R3 is “B”, R4 will be “C”**
- **R0 is zero, R1 is one**

End program with infinite loop

What should the control memory contents be?

A = A - B
If (A >= 0)
  C = C + 1
  goto start

	Add/ Sub	A_{R1}	A_{R2}	WE	A_W	N/Z/P	Next
0	1	R2	R3	1	R2	**000**	**X**
1	0	R2	R1	0	X	**100**	**#3**
2	1	R4	R1	1	R4	**111**	**#0**
3	X	X	X	0	X	**111**	**#3**

How might we improve our processing machine?

More operations & conditions

- **And, Not?**
- **Control next operation via any of negative/positive/zero combinations**

More data storage?

- **Add a separate data memory structure**
- **Register file used as “temporary” storage**
- **Add new logic elements to read and write this memory**

How would we sum all the numbers in memory?

- **Need “addressing modes” to allow this**
- **E.g., Read from the location in memory specified by R1**

Smaller encoding

- **Use fewer bits for “Next” (too large when control memory is big)**
- **Also want more “dynamic” control**
  - **E.g., next operation is at the location specified by R1’s value**

Warning!

This is a bottom-up course

- **No secrets, no magic**
  e.g., **gates build on transistors, logic circuits from gates, *etc*.**

But… some of this lecture is top-down

- **You'll have to trust me for a couple slides**
- **Start with *very* abstract discussion of computer architecture**
- **Meet with Chapter 3 material soon**

A Little Context

1943: ENIAC

- **First general electronic computer (Presper Eckert and John Mauchly)**
  (Or was it Atananasoff in 1939? Or Konrad Zuse in 1941?)
- **18,000 tubes (had to replace 50 a day!)**
- **Memory: 20 10-digit numbers (decimal)**
- **Hard-wired program (via dials, switches, and cables)**
- **Completed in 1946**

1944: Beginnings of EDVAC

- **Among other improvements, includes program stored in memory**
- **Gave birth to UNIVAC-I (1951)**
- **Completed in 1952**

Context Continued: Stored Program Computer

1945: John von Neumann

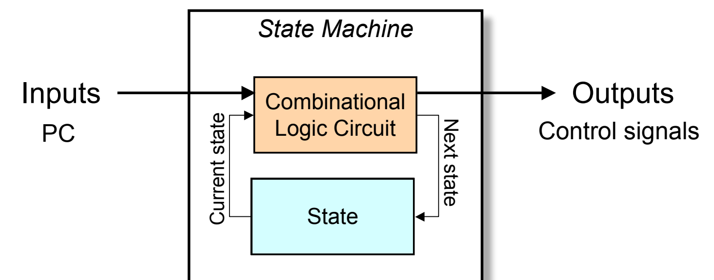
- ***First Draft of a Report on EDVAC***

See *John von Neumann and the Origins of Modern Computing* by William Aspray

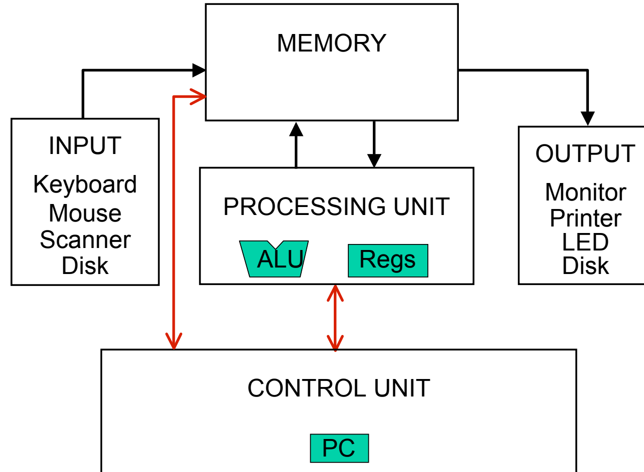
Von Neumann Machine (or Model)

- ***Memory*, containing instructions and data**
- ***Control unit*, for interpreting instructions**
- ***Processing unit*, for performing arithmetic and logical operations**
- ***Input/Output units*, for interacting with *real* world**

Remember Finite State Machines?



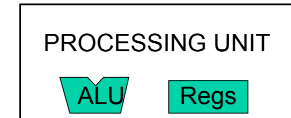
Von Neumann Model



Processing Unit

Functional Units

- **ALU = Arithmetic and Logic Unit**
- **Could have many functional units (some special-purpose, *e.g.,* multiply, square root, …)**
- **LC-3: ADD, AND, NOT**



Registers

- **Small, temporary storage**
- **Operands and results of functional units**
- **LC-3: eight register (R0, …, R7)**

Word Size

- **Number of bits normally processed by ALU in one instruction**
- **Also width of registers**
- **LC-3: 16 bits**

Memory

k x m array of stored bits (k is usually 2^n)

Address

- **Unique (*n*-bit) identifier of location**

Contents

- ***m*-bit value stored in location**

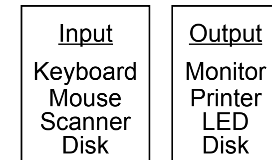
Basic Operations

- **Load: read a value from a memory location**
- **Store: write a value to a memory location**

Address	Contents
0000	
0001	
0010	
0011	00101101
0100	
0101	
0110	
	⋮
1101	10100010
1110	
1111	

Input and Output

Devices get data into and out of computer



Each device has own interface

- **LC-3 uses "memory-mapped registers"**
  - **Access with normal loads and stores**
- **LC-3 supports keyboard (input) and display (output)**
  - **Keyboard: data register (KBDR) and status register (KBSR)**
  - **Text display: data register (DDR) and status register (DSR)**
- **Graphical display: later…**

Some devices provide both input and output

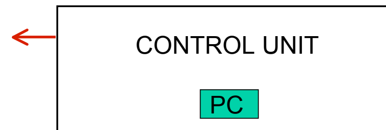
- **Disk, network**

Software that controls device access

- ***Driver***

Control Unit

Orchestrates execution of the program



Program Counter (PC)

- Contains the ***address*** of the next instruction to execute

Control Unit

- Reads an instruction from memory (at PC)
- Interprets the instruction
- Generates signals that tell the other components what to do
- Instruction may take many *machine cycles* to complete

Instructions

Fundamental unit of work

Constituents

- *Opcode*: operation to be performed
- *Operands*: data/locations to be used for operation

Encoded as a sequence of bits *(just like data!)*

- Sometimes have a fixed length (*e.g.*, 16 or 32 bits)
- Control unit interprets instruction
  - Generates control signals to carry out operation
- Atomic: operation is either executed completely, or not at all

Instruction Set Architecture (ISA)

- Computer's instructions, their formats, their behaviors

Instruction Set Architecture

ISA = *Programmer-visible* components & operations

- Memory organization
  - Address space -- how may locations can be addressed?
  - Addressibility -- how many bits per location?
- Register set
  - How many? What size? How are they used?
- Instruction set
  - Opcodes
  - Data types
  - Addressing modes

All information needed to write/gen machine language program

LC-3: Memory and Registers

Memory

- Address space: 2^{16} locations (16-bit addresses)
- Addressibility: 16 bits

Registers

- Temporary storage, accessed in a single machine cycle
  - Memory access generally takes longer
- Eight general-purpose registers: R0 - R7
  - Each 16 bits wide
  - How many bits to uniquely identify a register?
- Other registers
  - Not directly addressable, but used by (and affected by) instructions
  - PC (program counter), condition codes, *etc.*

LC-3: Instructions

Opcodes

- **16 opcodes**
- *Operate* **instructions: ADD, AND, NOT, (MUL)**
- *Data movement* **instructions: LD, LDI, LDR, LEA, ST, STR, STI**
- *Control* **instructions: BR, JSR, JSRR, RET, RTI, TRAP**
- **Some opcodes set/clear *condition codes*, based on result**
  - **N = negative (<0), Z = zero (=0), P = positive (> 0)**

Data Types

- **16-bit 2's complement integer**

Addressing Modes

- **How is the location of an operand specified?**
- **Non-memory addresses: *register*, *immediate (literal)***
- **Memory addresses: *base+offset*, *PC-relative*, *indirect***

LC-3 Instruction Summary

(inside back cover)

Instruction	15–12	Fields (bits 11–0, left to right)				
ADD+	0001	DR	SR1	0	00	SR2
ADD+	0001	DR	SR1	1	imm5	
AND+	0101	DR	SR1	0	00	SR2
AND+	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1	PCoffset11			
JSRR	0100	0	00	BaseR	000000	
LD+	0010	DR	PCoffset9			
LDI+	1010	DR	PCoffset9			
LDR+	0110	DR	BaseR	offset6		
LEA+	1110	DR	PCoffset9			
NOT+	1001	DR	SR	111111		
RET	1100	000	111	000000		
RTI	1000	000000000000				
ST	0011	SR	PCoffset9			
STI	1011	SR	PCoffset9			
STR	0111	SR	BaseR	offset6		
TRAP	1111	0000	trapvect8			
reserved	1101					

Example: LC-3 ADD Instruction

LC-3 has 16-bit instructions

- **Each instruction has a four-bit opcode, bits [15:12]**

LC-3 has eight *registers* (R0-R7) for temporary storage

- **Sources and destination of ADD are registers**

15–12	11–9	8–6	5	4	3	2–0
ADD	Dst	Src1	0	0	0	Src2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

"Add the contents of R2 to the contents of R6, and store the result in R6."

Example: LC-3 LDR Instruction

Reads data from memory

Base + offset addressing mode

- **Add offset to base register to produce memory address**
- **Load from memory address into destination register**

15–12	11–9	8–6	5–0
LDR	Dst	Base	Offset

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	0	0	0	1	1	0

"Add the value 6 to the contents of R3 to form a memory address. Load the contents of memory at that address and place the resulting data in R2."

Changing the Sequence of Instructions

Recall FETCH

- Increment PC by 1

What if we don't want linear execution?

- *E.g.,* loop, if-then, function call

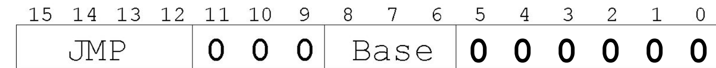
Need instructions that change PC

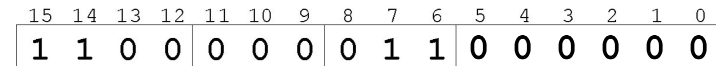
- Jumps are unconditional
  - Always change the PC
- Branches are conditional
  - Change the PC only if some condition is true
    e.g., the contents of a register is zero

Example: LC-3 JMP Instruction

Set the PC to the value of a register

- Fetch next instruction from this address





"Load the contents of register R3 into the PC."

LC-3: Operate Instructions

Only three operations

- ADD, AND, NOT, (MUL)

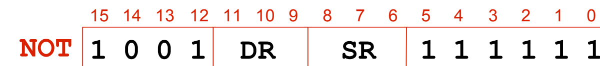
Source and destination operands are registers

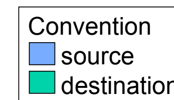
- ***Do not*** reference memory
- ADD and AND can use "immediate" mode,
  (*i.e.,* one operand is hard-wired into instruction)

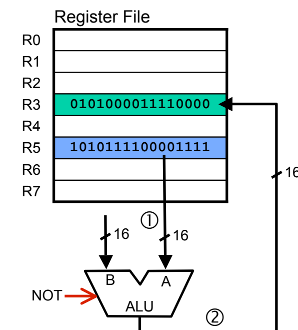
Will show abstracted datapath with each instruction

- Illustrate ***when*** and ***where*** data moves to accomplish desired op.

NOT (Register)



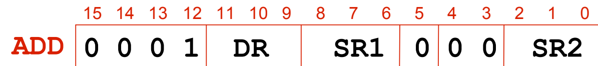


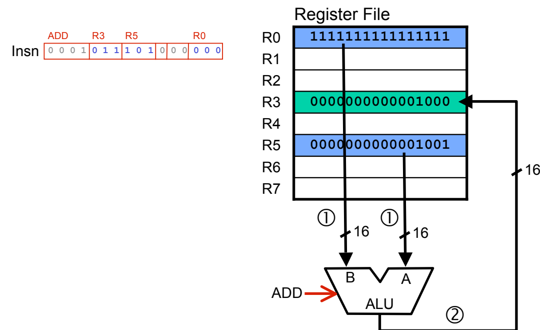


Note: DR and SR could be the <u>same</u> register

ADD (Register)

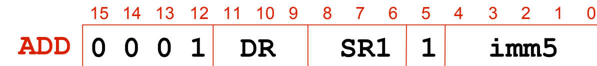
this zero means "register mode"

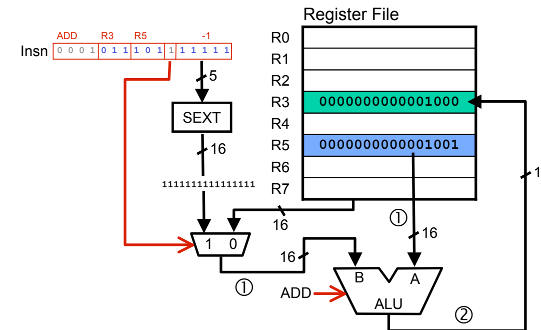




ADD (Immediate)

this one means "immediate mode"





SEXT = Sign Extension

Using Operate Instructions: Subtraction

How do we subtract two numbers?

Goal

- **R1 <- R2 - R3 (no such instruction in LC-3!)**

Idea (Use 2's complement)

1. **R1 <- NOT R3**
2. **R1 <- R1 + 1**
3. **R1 <- R2 + R1**

If 2nd operand is known and small, easy

- **R1 <- R2 + -3**

Using Operate Instructions: OR

How do we OR two numbers?

Goal

- **R1 <- R2 OR R3 (no such instruction in LC-3!)**

Idea (Use DeMorgan's Law)

- **A OR B = NOT(NOT(A) AND NOT(B))**

1. **R4 <- NOT R2**
2. **R5 <- NOT R3**
3. **R1 <- R4 AND R5**
4. **R5 <- NOT R1**

Using Operate Instructions: Copying

How do we copy a number from register to register?

Goal

- R1 <- R2   (no such instruction in LC-3!)

Idea (Use immediate)

- R1 <- R2 + 0

Could we use AND?

Using Operate Instructions: Clearing

How do we set a register to 0?

Goal

- R1 <- 0   (no such instruction in LC-3!)

Idea

- R1 <- R1 AND 0

Data Movement Instructions

Load: read data from memory to register

- LD: PC-relative mode
- LDR: base+offset mode
- LDI: indirect mode

Store: write data from register to memory

- ST: PC-relative mode
- STR: base+offset mode
- STI: indirect mode

Load effective address

- Compute address, save in register, do not access memory
- LEA: immediate mode

PC-Relative Addressing Mode

Want to specify address directly in the instruction

- But an address is 16 bits, and so is an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address

Observation

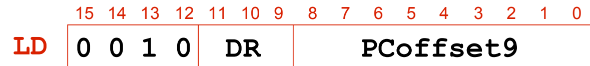
- Needed data often near currently executing instruction

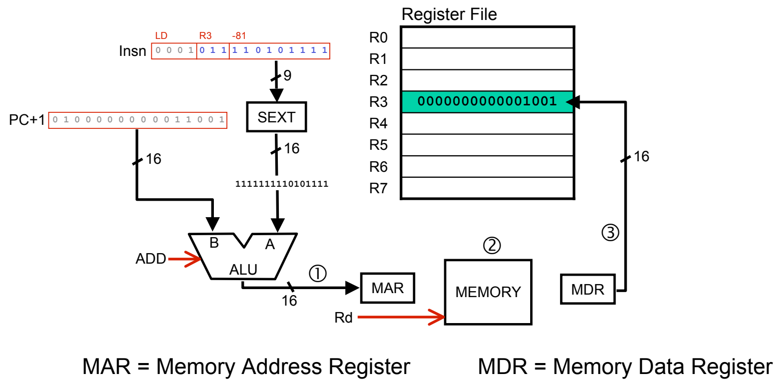
Solution

- Add 9 bits in instruction (sign extended) to PC (of *next instruction*) to form address

Example: LD: R1 <- Memory[PC+1 + SEXT(Insn[8:0])]

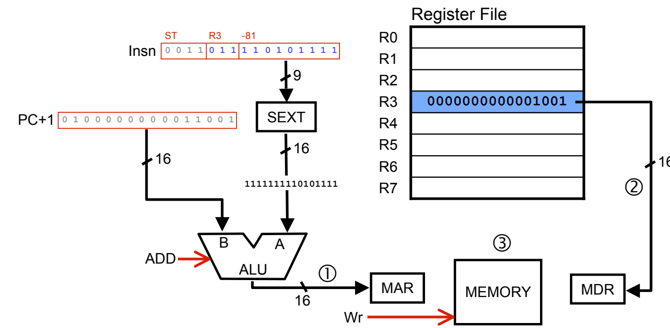
LD (PC-Relative)





MAR = Memory Address Register MDR = Memory Data Register

ST (PC-Relative)



Base + Offset Addressing Mode

Problem

- With PC-relative mode, can only address words “near” instruction
- What about the rest of memory?

Solution

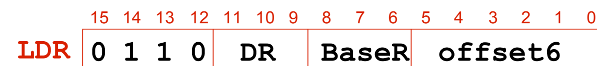
- Use a register to generate a full 16-bit address

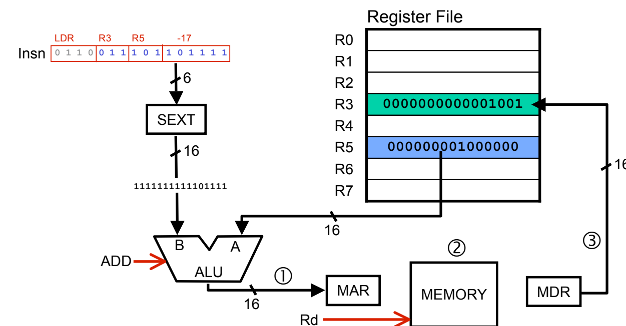
Idea

- 4 bits for opcode, 3 for src/dest register, 3 bits for *base* register
- Remaining 6 bits are used as a signed *offset*
- Offset is sign-extended before adding to base register
- *I.e.,* Instead of adding offset to PC, add it to base register

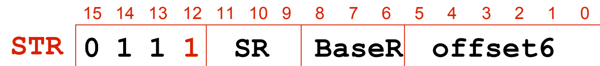
Example: LDR: R1 <- Memory[R2+SEXT(Insn[5:0])]

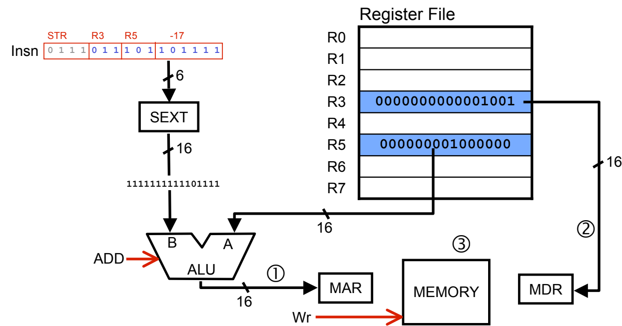
LDR (Base+Offset)





STR (Base+Offset)





Indirect Addressing Mode

Another way to produce full 16-bit address

- **Read address from memory, then load/store to that address**

Steps

- **Address is generated from PC and PCoffset**
  - **just like PC-relative addressing)**
- **Then content of that address is used as address for load/store**

Example: LDI: R1 <- Memory[Memory[PC+SEXT(Insn([8:0])]

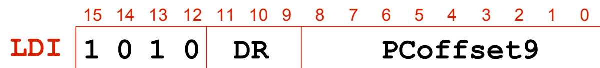
Advantage

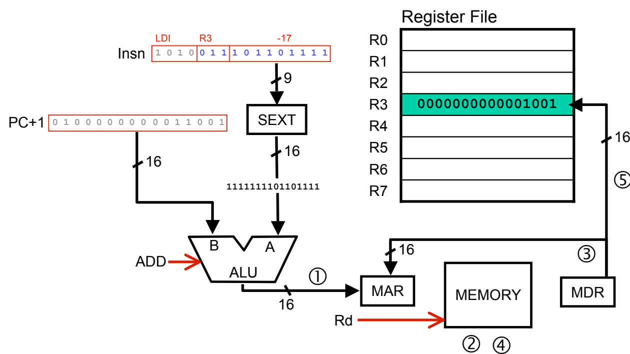
- **Doesn't consume a register for base address**
- **Addresses are often stored in memory (*i.e.,* useful)**

Disadvantage

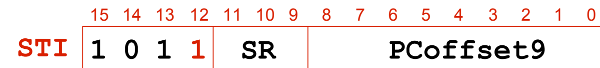
- **Extra memory operation (and no offset)**

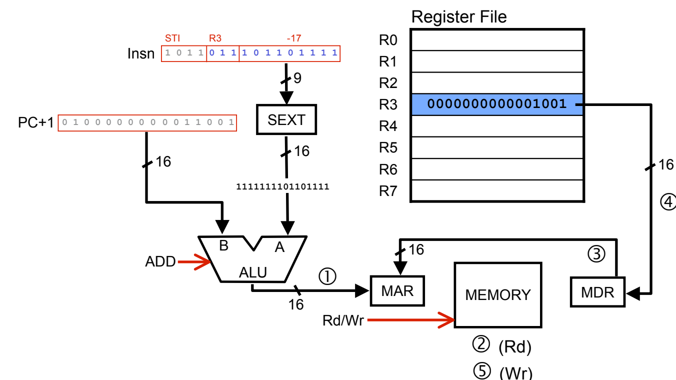
LDI (Indirect)





STI (Indirect)





Load Effective Address

Problem

- How can we compute address without also LD/ST-ing to it?

Solution

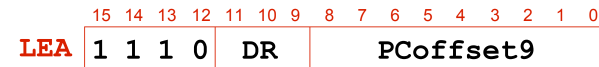
- Load Effective Address (LEA) instruction

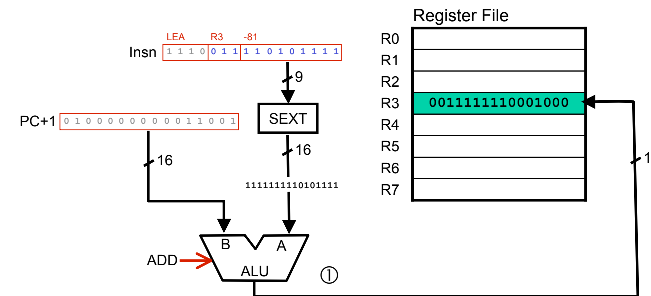
Idea

- LEA computes address just like PC-relative LD/ST
- Store address in destination register (not data at that address)
- Does not access memory

- Example: LEA: R1 <- PC + SEXT(Insn[8:0])]

LEA





Examples

Machine Language

0001	ADD
0011	ST
0101	AND
0111	STR
1010	LDI
1110	LEA

Address	opcode	*Instruction*	*Comments*
x30F6	1 1 1 0	0 0 1 1 1 1 1 1 1 1 0 1	R1 ← PC-3 (x30F4)
x30F7	0 0 0 1	0 1 0 0 0 1 1 0 1 1 1 0	R2 ← R1 + 14 (x3102)
x30F8	0 0 1 1	0 1 0 1 1 1 1 1 1 0 1 1	Mem[PC-5(x30F4)] ← R2
x30F9	0 1 0 1	0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x30FA	0 0 0 1	0 1 0 0 1 0 1 0 0 1 0 1	R2 ← R2 + 5
x30FB	0 1 1 1	0 1 0 0 0 1 0 0 1 1 1 0	Mem[R1+14] ← R2 (Mem[x3102] ← 5)
x30FC	1 0 1 0	0 1 1 1 1 1 1 1 0 1 1 1	R3 ← Mem[Mem[x30F4]] (R3 ← Mem[x3102]) (R3 ← 5)

opcode

Control Instructions

Alter the sequence of instructions

- Changing the Program Counter (PC)

Conditional Branch

- Branch *taken* if a specified condition is true
  - ➢ New PC computed relative to current PC
- Otherwise, branch *not taken*
  - ➢ PC is unchanged (I.e., points to next sequential instruction)

Unconditional Branch (or Jump)

- Always changes the PC
- Target address computed PC-relative or Base+Offset

Trap

- Changes PC to start of OS "service routine"
- When routine is done, execution resumes after TRAP instruction

Condition Codes

LC-3 has three 1-bit condition code registers

- **N -- negative**
- **Z -- zero**
- **P -- positive (greater than zero)**

Set/cleared by instructions that store value to register

- ***e.g.,* ADD, AND, NOT, LD, LDR, LDI, LEA (but *not* ST)**

Exactly <u>one</u> will be set at all times

- **Based on the last instruction that altered a register**

Branch Instruction

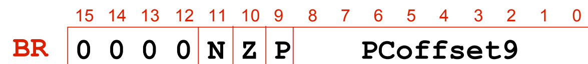
Branch specifies one or more condition codes

If the specified condition code set, the branch is taken

- **PC is set to the address specified in the instruction**
- **Like PC-relative mode addressing, target address is specified as offset from *next* PC (PC+1 + SEXT(Insn[8:0]))**
- **Note: Target must be "near" branch instruction**

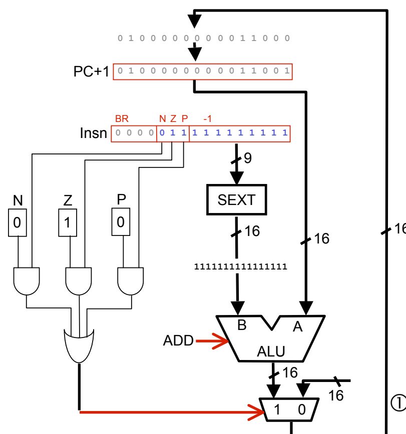
If branch not taken, next instruction (PC+1) is executed.

BR



Questions

- **Problems w/ this example?**
- **What if NZP all 0?**
- **What if NZP all 1?**



Example: Using Branch Instructions

Goal

- **Compute sum of 12 integers**

Input

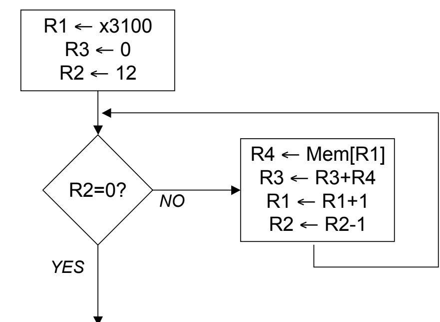
- **Numbers start at x3100**

Output

- **Register R3**

Program

- **Starts at x3000**



Example: Summing Program

0000	BR
0001	ADD
0110	LDR
0101	AND
1110	LEA

Address	*Instruction*	*Comments*
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	*R1 ← x3001+xFF (x3100)*
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	*R3 ← 0*
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	*R2 ← 0*
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	*R2 ← 12*
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1	*BRz x300A*
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	*R4 ← Mem[R1]*
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	*R3 ← R3+R4*
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	*R1 ← R1+1*
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	*R2 ← R2-1*
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	*BRnzp x3004*

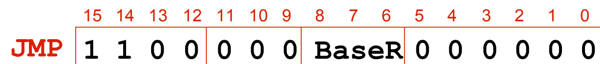
Jump Instructions

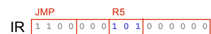
Jump is an unconditional branch (*i.e.*, *always* taken)

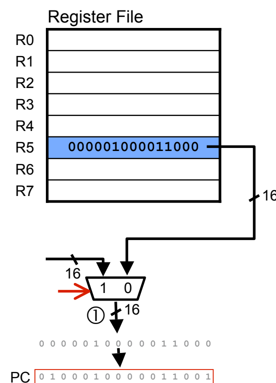
Destination

- **PC set to value of base register encoded in instruction**
- **Allows any branch target to be specified**
- **Pros/Cons versus BR?**

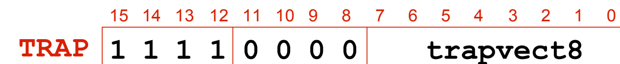
JMP







TRAP



Calls operating system "service routine"

- **Identified by unsigned 8-bit trap vector -- Zero Extension (ZEXT)**
- **Execution resumes after OS code executes (more later)**

vector	*routine*
x23	input a character from the keyboard
x21	output a character to the monitor
x25	halt the program (HALT)

Addressing Mode Summary

Register

- **R1 <- R1 + R2**
- **R1 <- NOT R2**

Immediate

- **R1 <- R1 + -2**

Base+Offset

- **R1 <- Mem[R2+4]**
- **Mem[R2+4] <- R1**

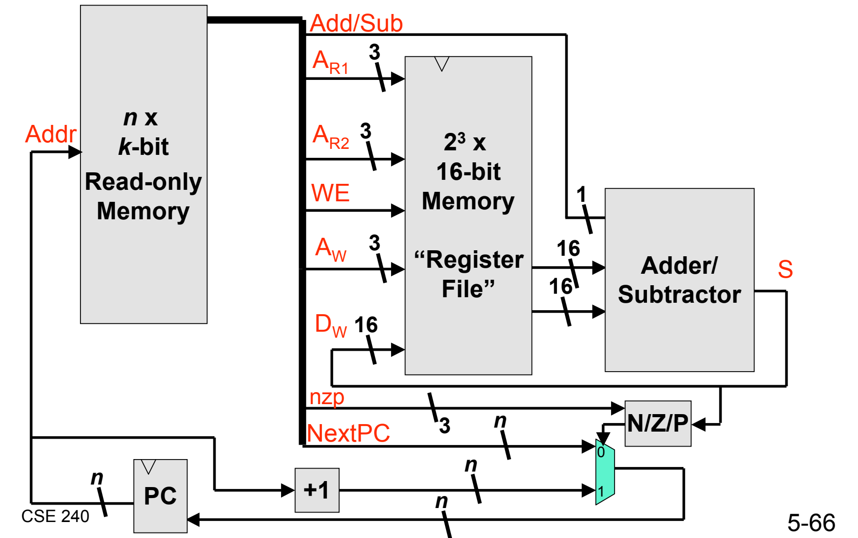
PC-Relative

- **R1 <- Mem[PC+6]**
- **Mem[PC+6] <- R1**

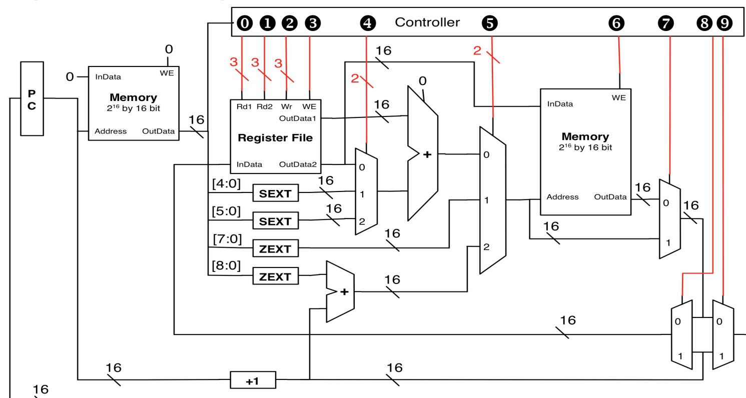
Indirect

- **R1 <- Mem[Mem[R2+4]]**
- **Mem[Mem[R2+4]] <- R1**

Remember this?



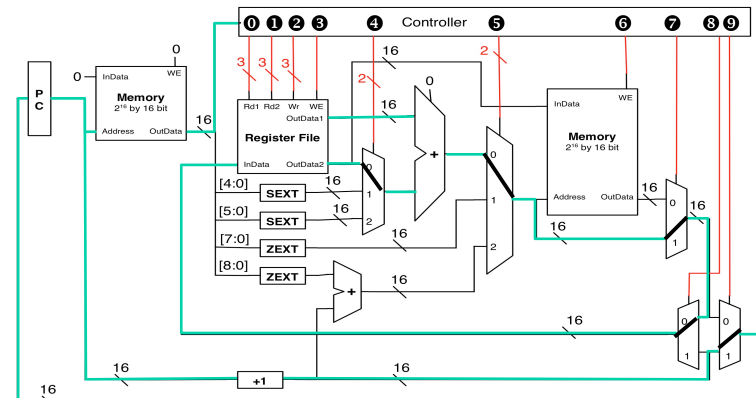
An (incomplete) LC-3 Implementation



Execute one instruction per cycle

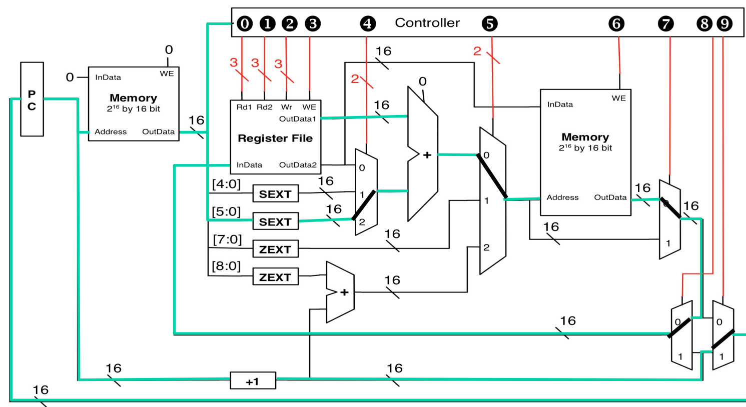
- **Much simpler than implementation in book**
- **All phases happen in one cycle**

ADD



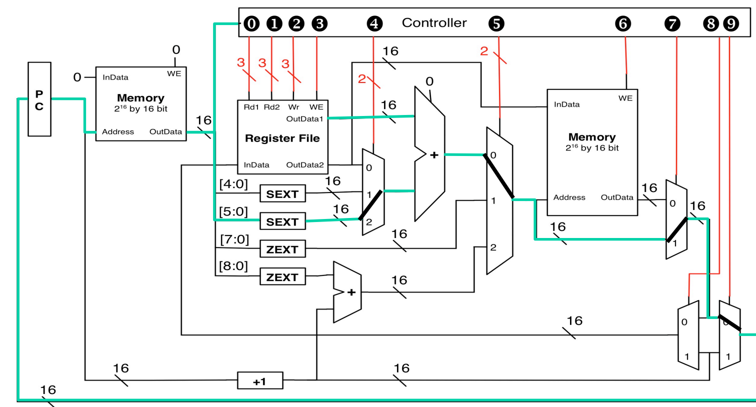
	Opcode		Control									
Instr	I[15:12]	I[5]	⓿	❶	❷	❸	❹	❺	❻	❼	❽	❾
ADD	0001	0	I[8:6]	I[2:0]	I[11:9]	1	00	00	0	1	0	1

LDR



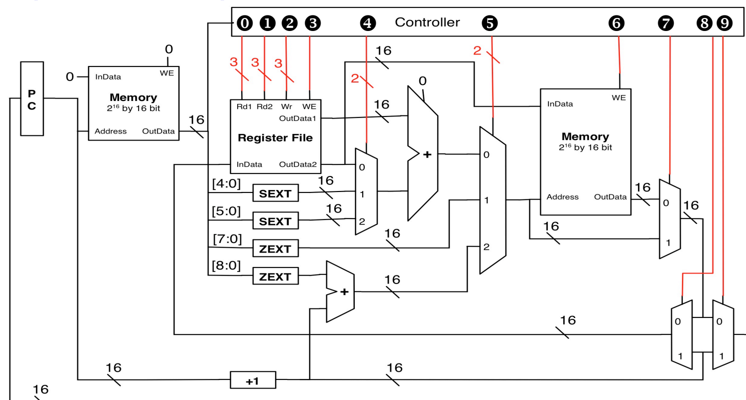
	Opcode		Control									
Instr	I[15:12]	I[5]	⓿	❶	❷	❸	❹	❺	❻	❼	❽	❾
LDR	0110	-	I[8:6]	-	I[11:9]	1	2	00	0	0	0	1

JMP



	Opcode		Control									
Instr	I[15:12]	I[5]	⓿	❶	❷	❸	❹	❺	❻	❼	❽	❾
JMP	1100	-	I[8:6]	-	-	0	2	00	0	1	x	0

An (incomplete) LC-3 Implementation



AND? NOT? What changes would you make?
(LDI/STI? JSR? RTI?)

Another Example

Count the occurrences of a character in a file

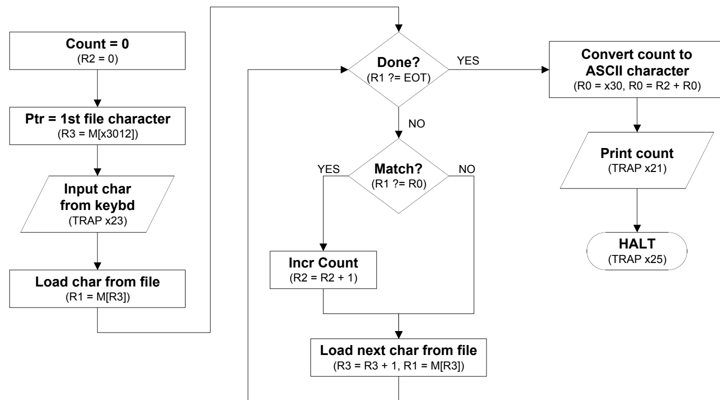
- **Program begins at location x3000**
- **Read character from keyboard**
- **Load each character from a "file"**
  - File is a sequence of memory locations
  - Starting address of file is stored in the memory location immediately after the program
- **If file character equals input character, increment counter**
- **End of file is indicated by a special ASCII value: EOT (x04)**
- **At the end, print the number of characters and halt**
  (assume there will be fewer than 10 occurrences of the character)

A special character used to indicate the end of a sequence is often called a sentinel

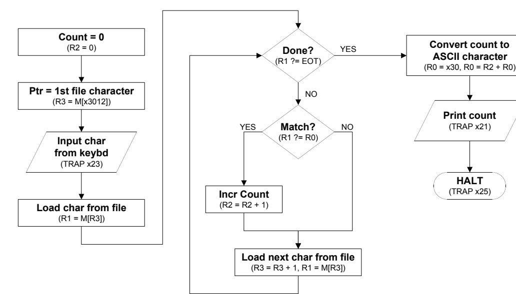
- Useful when you don't know ahead of time how many times to execute a loop

Flow Chart

Input: Mem[x3012] (address of "file")
Output: Print count to display



Program



R2 ← 0 (Count)
R3 ← Mem[x3012] (Ptr)
Input to R0 (TRAP x23)
R1 ← Mem[R3]
R4 ← R1 – 4 (EOT)
BRz x????
R1 ← NOT R1
R1 ← R1 + 1
R1 ← R1 + R0
BRnp x????
R2 ← R2 + 1
R3 ← R3 + 1
R1 ← Mem[R3]
BRnzp x????
R0 ← Mem[x3013]
R0 ← R0 + R2
Print R0 (TRAP x21)
HALT (TRAP x25)

Program (1 of 2)

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

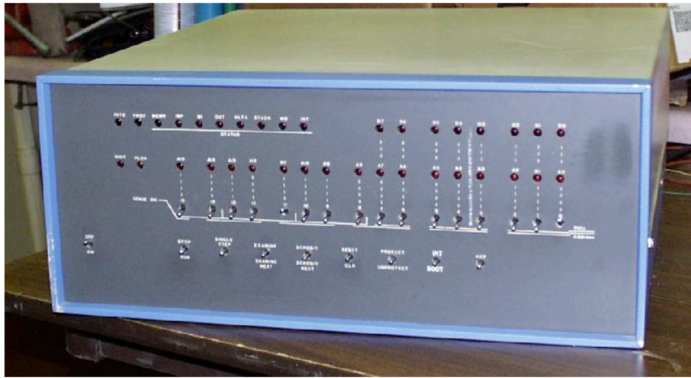
Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0 (counter)
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	R3 ← M[x3012] (ptr)
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	Input to R0 (TRAP x23)
x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3]
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	R4 ← R1 – 4 (EOT)
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	BRz x300E
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	R1 ← NOT R1
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	R1 ← R1 + 1
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	R1 ← R1 + R0
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	BRnp x300B

Program (2 of 2)

0000	BR
0001	ADD
0010	LD
0101	AND
1111	TRAP

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	R2 ← R2 + 1
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	R3 ← R3 + 1
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3]
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	BRnzp x3004
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	R0 ← M[x3013]
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0	R0 ← R0 + R2
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	Print R0 (TRAP x21)
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	HALT (TRAP x25)
X3012	Starting Address of File	
x3013	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0	ASCII x30 ('0')

Aside: Machine Language Programming Is Hard!



(Altair 8800, 1975)

Summary

Many instructions

- ISA: Programming-visible components and operations
- Behavior determined by opcodes and operands
  - ➢Operate, Data, Control
- Control unit "tells" rest of system what to do (based on opcode)
- Some operations must be synthesized from given operations (*e.g.,* subtraction, logical or, *etc.*)

Concepts

- Addressing modes
- Condition codes and branching/jumping

Bit-level programming bites!