# Chapter 5
## The LC-3

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

## 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/generate machine language program



## Instruction

**Fundamental unit of work**

**Constituents**

- ***Opcode***: operation to be performed (e.g. add, and)
- ***Operands***: data/locations to be used for operation
  - Source: location that contains the data/instruction
  - Destination: location that will store the result of computation
  - Immediate: data values not contained at a particular location



## LC-3 Overview: Memory and Registers

**Memory**

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

**Registers**

- Temporary storage (Memory access generally takes longer)
- Eight general-purpose registers: R0 - R7 (each 16 bits wide)
- Other registers
  - Not directly addressable, but used by (and affected by) instructions
  - PC (program counter), condition codes, MAR, MDR, *etc.*

**Word Size**

- 16 bits

# LC-3 ISA: Overview

**Opcodes**

- 16 opcodes ([15:12] of instruction = $2^4$ = 16 possible values)
- Types of instructions:
  - *Operate* instructions: ADD, AND, NOT
  - *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
  - *Control* instructions: BR, JUMP, TRAP JSR, JSRR, RET, RTI
- Some opcodes set/clear *condition codes*, based on result
  - N = negative (<0), Z = zero (=0), P = positive (> 0)

**Addressing Modes**

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

**Data Types**

- 16-bit 2's complement integer



# Operate Instructions

**Only three operations**

- ADD, AND, NOT

**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.

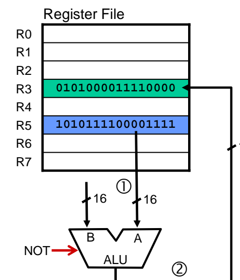


# NOT (Register format)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NOT | 1 | 0 | 0 | 1 | DR | SR | 1 | 1 | 1 | 1 | 1 | 1 |

IR: NOT R3 R5 — 1001 011 101 111111

Convention: source (blue), destination (green)




*Note: DR and SR could be the same register*



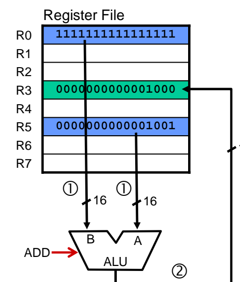
# ADD (Register format)

*this zero means "register mode"*

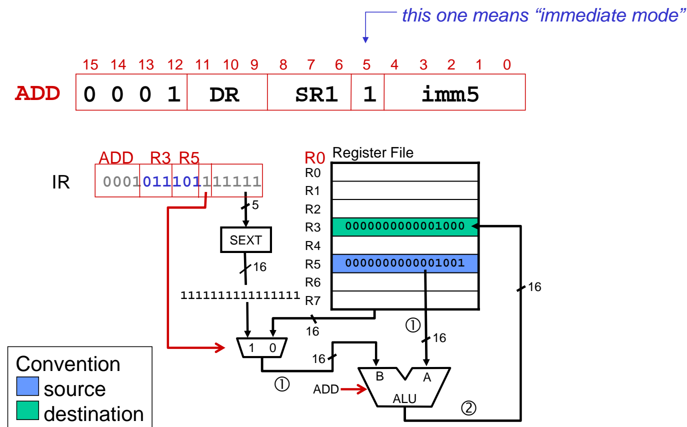
| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 | 5 | 4 | 3 | 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | SR1 | 0 | 0 | 0 | SR2 |

IR: ADD R3 R5 R0 — 0001 011 101 0 00 000

Convention: source (blue), destination (green)



## ADD (Immediate format)






## Using Operate Instructions: Subtraction

**How do we subtract two numbers?**

**Goal**

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

**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!)**

**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!)**

**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!)

**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

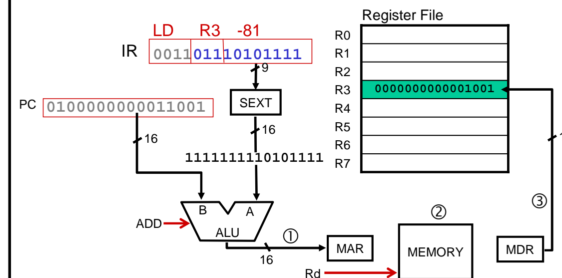
**Solution**

- Take 9 bits [8:0] in instruction
- Sign extended to 16 bits
- Add it to the PC (of *next instruction*) to form address



## LD (PC-Relative)

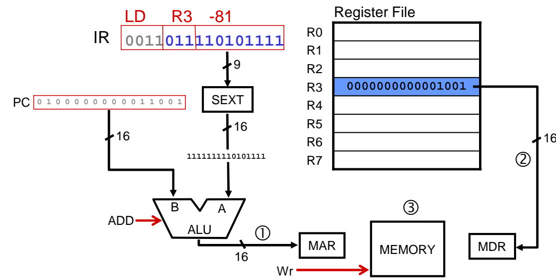
| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| **LD** | 0 | 0 | 1 | 0 | DR | PCoffset9 |




**Example: LD: R1 <- M[PC+SEXT(IR[8:0])]**

## ST (PC-Relative)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | PCoffset9 |

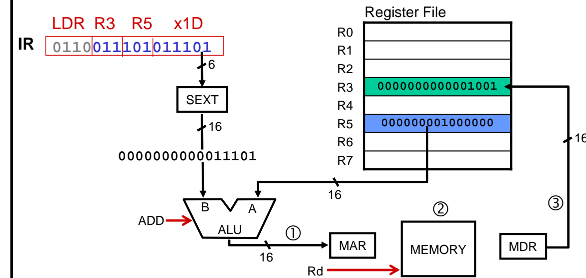



CIT 593

5-17

## LDR (Base+Offset)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 | 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| LDR | 0 | 1 | 1 | 0 | DR | BaseR | offset6 |




CIT 593

5-19

## Base + Offset Addressing Mode

**Problem**
- With PC-relative mode, can only address words "near" the 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

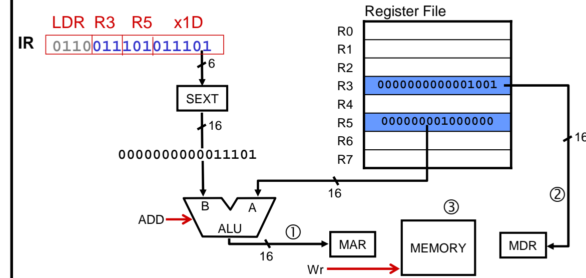
**Example: LDR: R1 <- M[R2+SEXT(IR[5:0])]**

CIT 593

5-18

## STR (Base+Offset)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 | 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|
| STR | 0 | 1 | 1 | 1 | SR | BaseR | offset6 |




CIT 593

5-20

# Indirect Addressing Mode

**Another way to produce full 16-bit address**

- Read address from memory location, 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 <- M[M[PC+SEXT(IR([8:0])]**

**Advantage**

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

**Disadvantage**

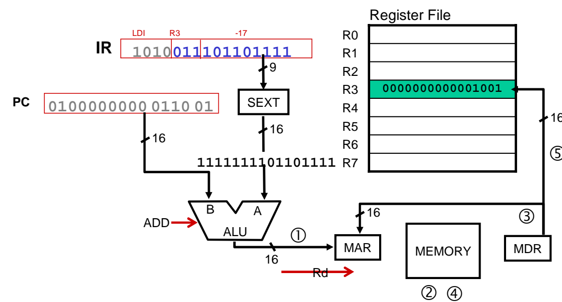
- Extra memory operation (and no offset)

CIT 593



# LDI (Indirect)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| LDI | 1 | 0 | 1 | 0 | DR | PCoffset9 |

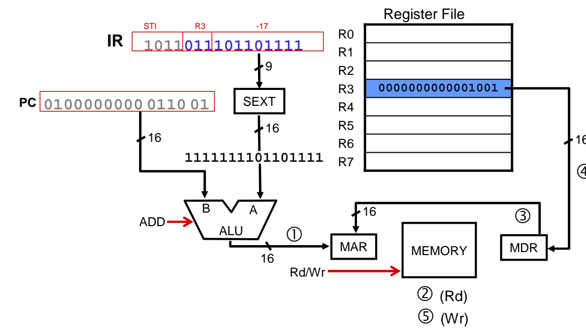



CIT 593



# STI (Indirect)

| | 15 | 14 | 13 | 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|
| STI | 1 | 0 | 1 | 1 | SR | PCoffset9 |




CIT 593



# Load Effective Address

**Problem**

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

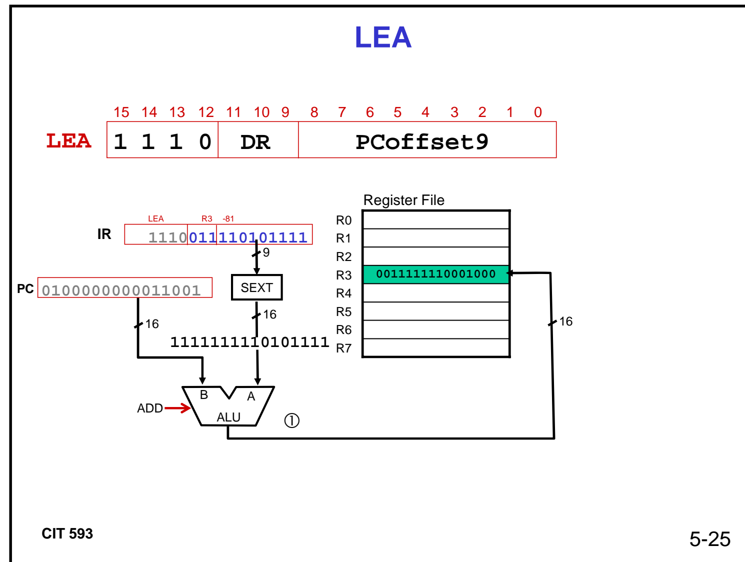
**Solution**

- Load Effective Address (LEA) instruction
- Used to generate address
  - Initializes a register with an address very close to the initializing 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(IR[8:0])]

CIT 593




## Example

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 1 0 1 | 0 1 0 0 1 0 1 0 0 0 0 0 | R2 ← 0 |
| x30F8 | 0 0 0 1 | 0 1 0 0 1 0 1 0 0 1 0 1 | R2 ← R2 + 5 = 5 |
| x30F9 | 0 1 1 1 | 0 1 0 0 0 1 0 0 1 1 1 0 | M[R1+14] ← R2 (M[x3102] ← 5) |

Example: stores the value 5 at particular location

Example: 5.10 (pg 129)
Shortened version



## 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



## 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, *not* ST

Exactly one will be set at all times
- Based on the last instruction that altered a register

## Branch Instruction

**Branch specifies one or more condition codes**

- N, Z, or P

**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 current PC (PC + SEXT(IR[8:0]))
- Note: Target must be "near" branch instruction

**If branch not taken i.e. the condition is not met, then next instruction (PC+1) is executed.**

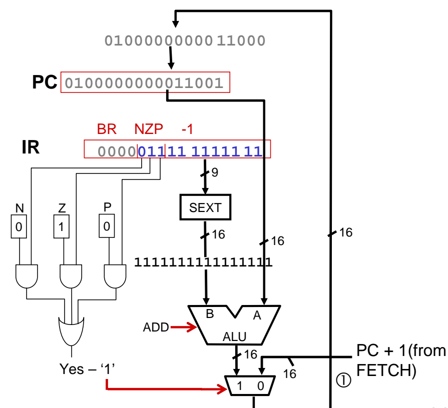


## BR

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | N | Z | P | PCoffset9 | | | | | | | | |

**Questions**

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






## Jump Instructions

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

**So why have an special Jump instruction when we can do the same with BR?**

➤ Because with BR we can move only in the range of the offset

**Destination**

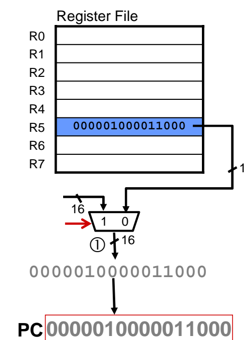
- PC set to value of base register encoded in instruction
- Allows any branch target to be specified



## JMP

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

IR: JMP R5 — 1100000 101 000000



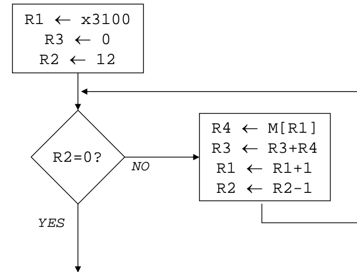
## Example: Using Branch Instructions

### Goal
- Compute sum of 12 integers

### Input
- Numbers start at x3100

### Output
- Register R3 (contains sum)

### Program
- Starts at x3000




CIT 593



## 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 ← M[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 |
| x300A | | |

CIT 593



## TRAP

| | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 3 2 1 0 |
|---|---|---|---|
| TRAP | 1 1 1 1 | 0 0 0 0 | trapvect8 |

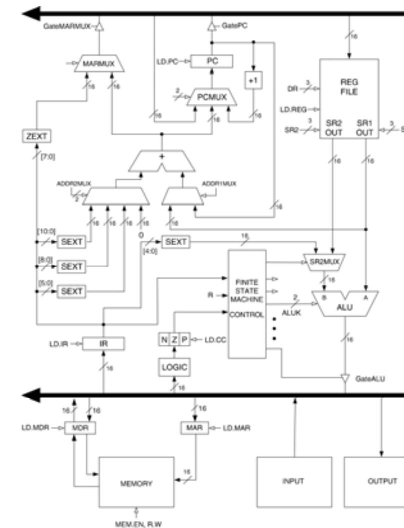
### Calls operating system “service routine”
- Identified by 8-bit trap vector
- Execution resumes after OS code executes (more later)

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

CIT 593



## LC-3 Data Path Revisited



**Filled arrow**
= info to be processed.
**Unfilled arrow**
= control signal.

CIT 593

## Data Path Components

**Global bus**

- Set of wires that carry 16-bit signals to many components
- Inputs to bus are controlled by triangle structure called tri-state devices (just remember the triangle for now)
  - Place signal on bus when enabled
  - Only one (16-bit) signal should be enabled at a time
  - Control unit decides which signal "drives" the bus
- Any number of components can read bus
  - Register only captures bus data if write-enabled by the control unit

**Memory and I/O**

- Control and data registers for memory and I/O devices
- Memory: MAR, MDR (also control signal for read/write)
- Input (keyboard): KBSR, KBDR
- Output (text display): DSR, DDR



## Data Path Components (cont.)

**ALU**

- Input: register file or sign-extended bits from IR (immediate field)
- Output: bus; used by…
  - Condition code registers
  - Register file
  - Memory and I/O registers

**Register File**

- Two read addresses, one write address (3 bits each)
- Input: 16 bits from bus
  - Result of ALU operation or memory (or I/O) read
- Outputs: two 16-bit
  - Used by ALU, PC, memory address
  - Data for store instructions passes through ALU



## Data Path Components (cont.)

**PC and PCMUX**

- Three inputs to PC, controlled by PCMUX
  1. Current PC plus 1 (normal operation)
  2. Adder output (BR, JMP, …)
  3. Bus (TRAP)

**MAR and MARMUX**

- Some inputs to MAR, controlled by MARMUX
  1. Zero-extended IR[7:0] (used for TRAP; more later)
  2. Adder output (LD, ST, …)



## Data Path Components (cont.)

**Condition Code Logic**

- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them
  - Only certain instructions set the codes
    (anything that places a value into a register:
    ADD, AND, NOT, LD, LDI, LDR, LEA, not ST)

**Control Unit**

- Decodes instruction (in IR)
- On each machine cycle, changes control signals for next phase of instruction processing
  - Who drives the bus?
  - Which registers are write enabled?
  - Which operation should ALU perform?
  - …

## 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!**



## LC-3 Instruction Summary
(inside back cover)

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



## Next Time

**1-2 Examples from chapter 5 & 6**

**Chapter 7: Assembly Language Program**

**Reading: Chp6**
**Chp 7**



## 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:** M[x3012] (address of "file")
**Output:** Print count to display

Count = 0
(R2 = 0)
Ptr = 1st file character
(R3 = M[x3012])
Input char from keybd
(TRAP x23)
Load char from file
(R1 = M[R3])
Done?
(R1 ?= EOT)
YES
Convert count to ASCII character
(R0 = x30, R0 = R2 + R0)
Print count
(TRAP x21)
HALT
(TRAP x25)
NO
Match?
(R1 ?= R0)
YES
Incr Count
(R2 = R2 + 1)
NO
Load next char from file
(R3 = R3 + 1, R1 = M[R3])



## Program

Count = 0
(R2 = 0)
Ptr = 1st file character
(R3 = M[x3012])
Input char from keybd
(TRAP x23)
Load char from file
(R1 = M[R3])
Done?
(R1 ?= EOT)
YES
Convert count to ASCII character
(R0 = x30, R0 = R2 + R0)
Print count
(TRAP x21)
HALT
(TRAP x25)
NO
Match?
(R1 ?= R0)
YES
NO
Incr Count
(R2 = R2 + 1)
Load next char from file
(R3 = R3 + 1, R1 = M[R3])

*R2 ← 0 (Count)*
*R3 ← M[x3012] (Ptr)*
*Input to R0 (TRAP x23)*
*R1 ← M[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 ← M[R3]*
*BRnzp x????*
*R0 ← M[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')* |