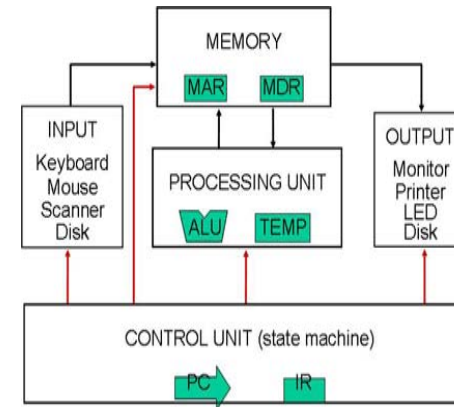


Chapter 5

The LC-3

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

Recap: Von Nuemann Model



CIT 593

2/41

ISA Recap

All information needed to write/generate machine language program

- Memory Organization
- Register Set
- Instruction Set

Instruction: Fundamental unit of work

- **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

CIT 593

3/41

LC-3 Overview: Memory and Registers

Memory

- Address space: 2^{16} locations (16-bit addresses)
- Addressability: 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), CC (condition codes), MAR, MDR, etc.

Word Size

- 16 bits

CIT 593

4/41

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/RTT, RTI
- Some opcodes set/clear CC (*condition codes*), based on result

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

CIT 593

5/41

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.

CIT 593

6/41

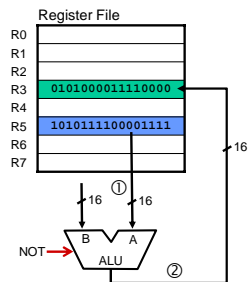
NOT (Register format)

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
 1 001 011 101 111111

Convention
 ■ source
 ■ destination

Note: DR and SR could be the same register



CIT 593

7/41

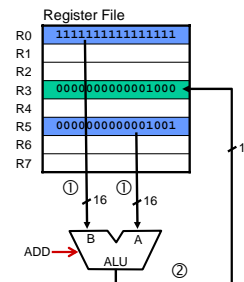
ADD (Register format)

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

this zero means "register mode"

IR ADD R3 R5 R0
 0001 011 101 000000

Convention
 ■ source
 ■ destination



CIT 593

8/41

Using Operate Instructions: Copying

How do we copy a number from register to register?

Goal

- $R1 \leftarrow R2$ (no such instruction!)

Idea (Use immediate)

- $R1 \leftarrow R2 + 0$

Could we use AND?

R1 ← R2 AND ?															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CIT 593

13/41

Using Operate Instructions: Clearing

How do we set a register to 0?

Goal

- $R1 \leftarrow 0$ (no such instruction!)

Idea

- $R1 \leftarrow R1 \text{ AND } 0$

R1 ← R2 AND 0															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CIT 593

14/41

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

CIT 593

15/41

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
 - Because once the instruction is Fetched from memory at location indicated by PC, PC is incremented by 1 (default)

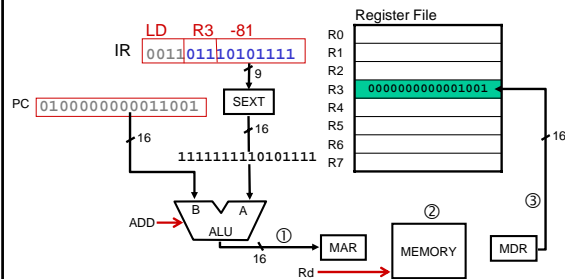
Limitations

- What is the max range of addresses we span from PC ?

CIT 593

16/41

LD (PC-Relative)



Example: LD: R1 ← M[PC+SEXT(IR[8:0])]

CIT 593

17/41

LC-3 Instruction Encoding: Example 2

1. We want to write an instruction that will load contents from memory location x3025 into register R2
2. This instruction will be placed in memory at location x3004

Is this achievable using LD?

If so write the instruction encoding

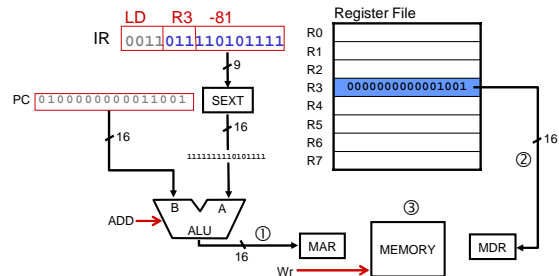
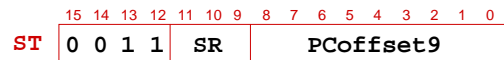
R2 ← M[3025]

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

CIT 593

18/41

ST (PC-Relative)



CIT 593

19/41

Load Effective Address

Problem

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

Solution

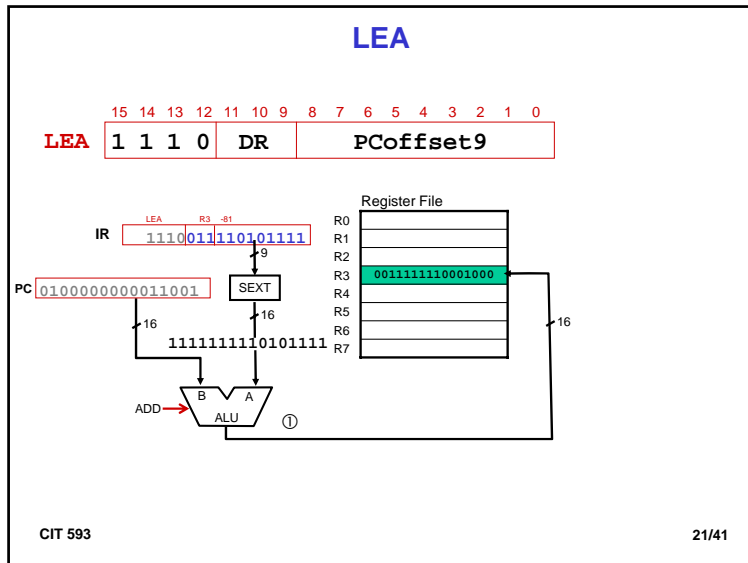
- Load Effective Address (LEA) instruction
- Used to generate address
 - Initializes a register with an address very close to the initializing instruction

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

20/41



Base + Offset Addressing Mode

Problem

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

Solution

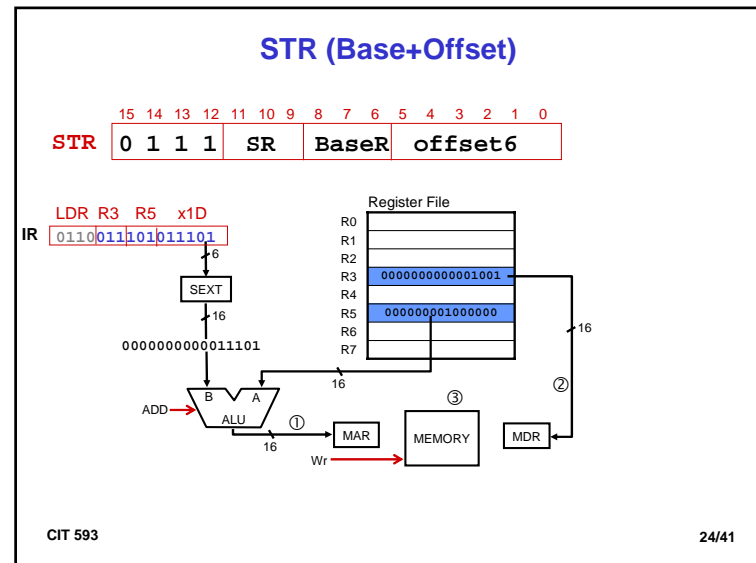
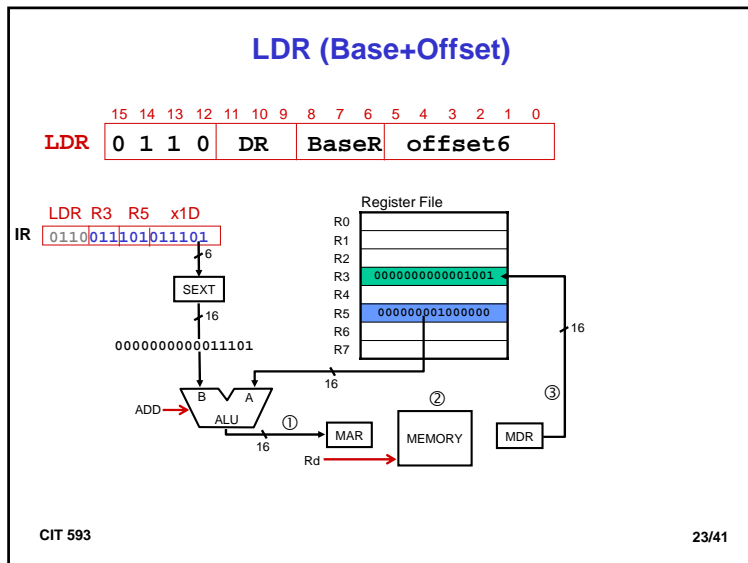
- Use a register to generate a full 16-bit address

Idea

- 4 bits for opcode, 3 for src/dest register, 3 bits for **base** register
 - Base Register is setup using LEA instruction
- 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

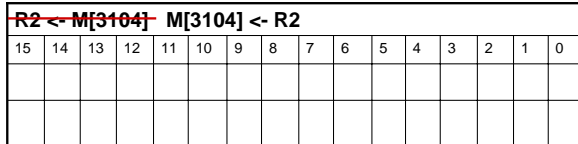
Example: LDR: R1 ← M[R2+SEXT(IR[5:0])]

CIT 593 22/41



LC-3 Instruction Encoding: Example 3

1. We write an instruction that will store contents from R2 into memory at location x3104
2. This instruction will be placed in memory at location x3000



CIT 593

25/41

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+SXT(IR[8:0])]]

Advantage

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

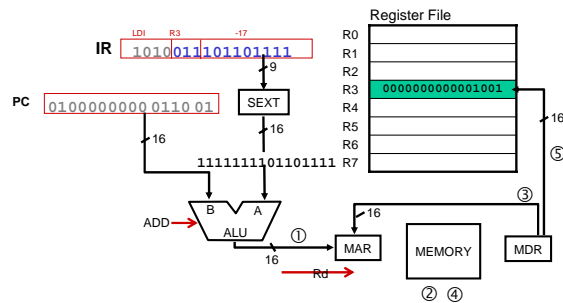
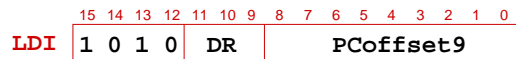
Disadvantage

- Extra memory operation

CIT 593

26/41

LDI (Indirect)

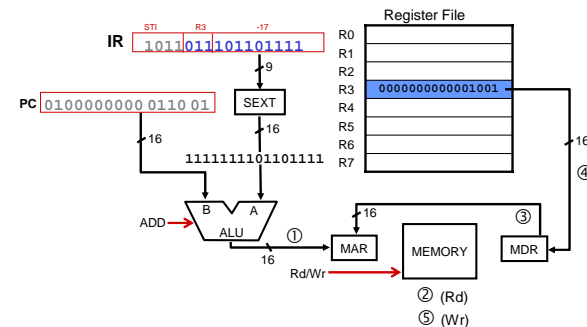
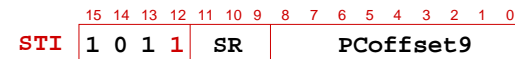


CIT 593

Can we deal without such an instruction ?

27/41

STI (Indirect)



CIT 593

28/41

Control Instructions

Altering sequence of instructions

- PC is always incremented in FETCH stage
 - By default we always move to next instruction
- An instruction can cause alter the flow of the program
 - Then PC + 1 value will be overridden

1. 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)

2. Unconditional Branch (or Jump)

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

3. TRAP

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

CIT 593

29/41

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,
- Note store (ST,STR,STI) instructions do not update the NZP registers

Exactly one will be set at all times

- Based on the last instruction that altered a register

CIT 593

30/41

Branch Instruction

Branch specifies one or more condition codes

- N, Z, or P

If the specified condition code set, the branch is taken

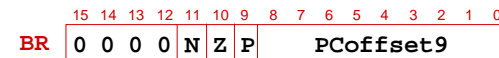
- PC is set to the address specified in the instruction
- Like PC-relative mode addressing, **target address** is specified as offset from current PC (PC + SEXT(IR[8:0]))
- Note: Target must be "near" branch instruction

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

CIT 593

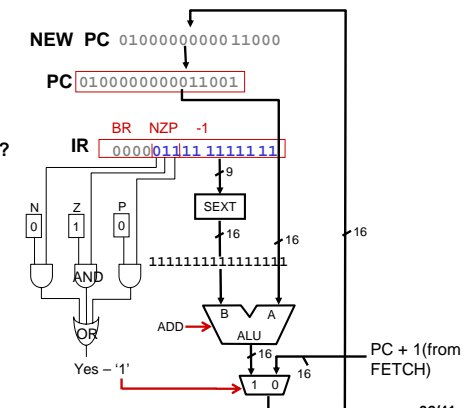
31/41

BR



Questions

- Problems w/ this example?
 - Infinite loop
- What if NZP all 0?
 - Next instr = PC + 1
- What if NZP all 1?
 - Always Branch



CIT 593

32/41

LC-3 Instruction Encoding: Example 4

Give the encoding of a Branch instruction located at x3009 that always changes PC to x3005

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

CIT 593

33/41

Example: Using Branch Instructions

Goal

- Compute **sum** of 12 integers

Input

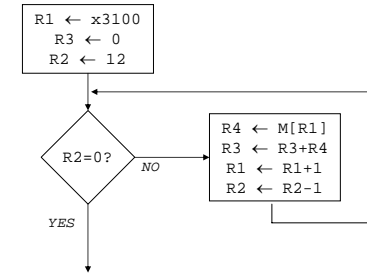
- Numbers start at x3100

Output

- Register R3 (contains **sum**)

Program

- Starts at x3000



CIT 593

34/41

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	$R1 \leftarrow x3001 + xFF (= x3100)$
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	$R3 \leftarrow 0$
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	$R2 \leftarrow 0$
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	$R2 \leftarrow 12$
x3004	0 0 0 0 0 1 0 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	$R4 \leftarrow M[R1]$
x3006	0 0 0 1 0 1 1 0 0 1 1 0 0 0 1 0 0	$R3 \leftarrow R3 + R4$
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 0 1	$R1 \leftarrow R1 + 1$
x3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	$R2 \leftarrow R2 - 1$
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	$BRnzp\ x3004$

CIT 593

35/41

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?

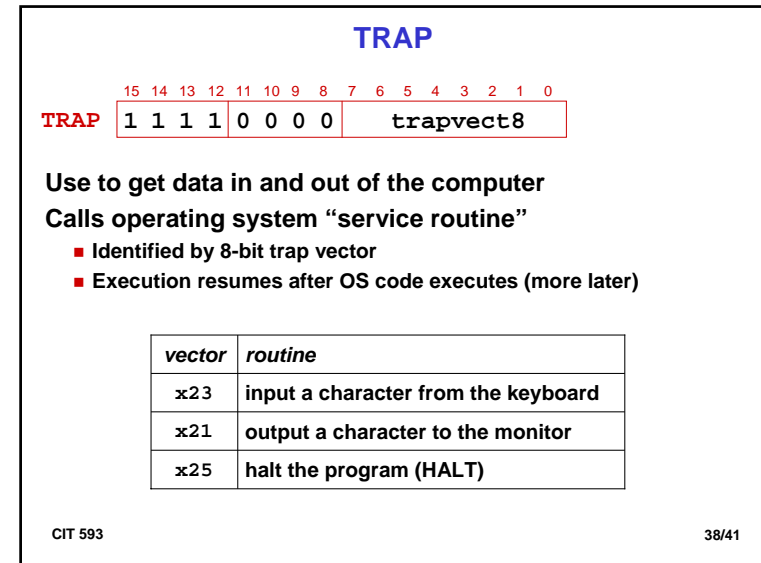
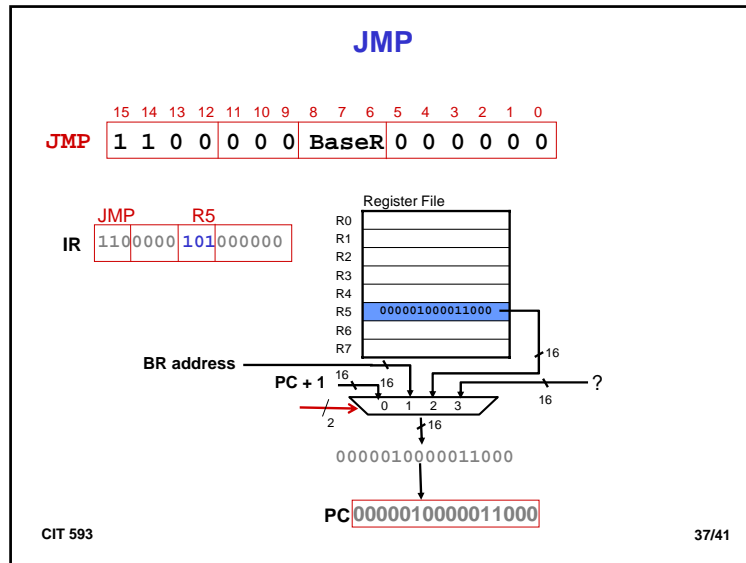
- Because with BR we can move only in the range of the offset

Destination

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

CIT 593

36/41



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
LDR*	1010	DR														PCoffset9
LDR*	0110	DR	BaseR													offset5
LEA*	1110	DR														PCoffset9
NOT*	1001	DR	SR													111111
RET	1100	000														000000
RTI	1000															000000000000
ST	0011	SR														PCoffset9
STI	1011	SR														PCoffset9
STR	0111	SR	BaseR													offset5
TRAP	1111	0000														trapvect8
reserved	1101															

CIT 593 39/41

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!

CIT 593 40/41

Next Time

1-2 Examples from chapter 5

Chapter 7: Assembly Language Program

Reading: Chp6 (will not cover in class – Please read for learning good procedural programming & debugging techniques)