

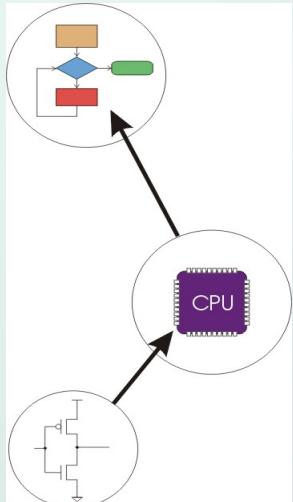
Chapter 5 The LC-3

Original slides from Gregory Byrd, North Carolina State University

Modified slides by C. Wilcox, S. Rajopadhye
Colorado State University

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Computing Layers



Problems

Algorithms

Language

Instruction Set Architecture 

Microarchitecture

Circuits

Devices

Instruction Set Architecture

- ISA = All of the **programmer-visible** components and operations of the computer
 - **memory organization**
 - address space -- how many locations can be addressed?
 - addressability -- how many bits per location?
 - **register set**
 - how many? what size? how are they used?
 - **instruction set**
 - opcodes
 - data types
 - addressing modes
- ISA provides all information needed for someone that wants to write a program in **machine language**
 - or translate from a high-level language to machine language.

LC-3 Overview: Memory and Registers

- **Memory**
 - address space: 2^{16} locations (16-bit addresses)
 - addressability: **16 bits**
- **Registers**
 - temporary storage, accessed in a single machine cycle
 - accessing memory takes longer than a single cycle
 - 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**

LC-3 Overview: Instruction Set

• **Opcodes**

- 15 opcodes, 3 types of instructions
- **Operate**: ADD, AND, NOT
- **Data movement**: LD, LDI, LDR, LEA, ST, STR, STI
- **Control**: BR, JSR/JSRR, JMP, RTI, TRAP
- some opcodes set/clear *condition codes*, based on result:
 - N = negative, Z = zero, P = positive (> 0)

• **Data Types**

- 16-bit 2's complement integer

• **Addressing Modes**

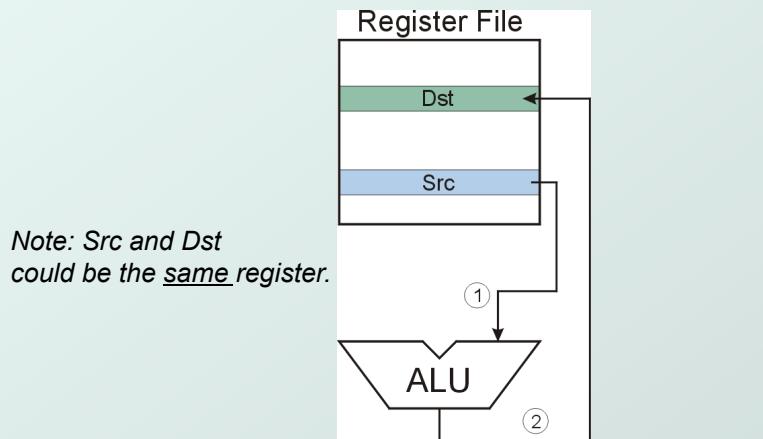
- How is the location of an operand specified?
- non-memory addresses: *immediate, register*
- memory addresses: *PC-relative, indirect, base+offset*

Operate Instructions

- Only three operations: **ADD, AND, NOT**
- Source and destination operands are **registers**
 - These instructions do not reference memory.
 - ADD and AND can use “immediate” mode, where one operand is hard-wired into the instruction.
- Will show **dataflow diagram** with each instruction.
 - illustrates when and where data moves to accomplish the desired operation

NOT (Register)

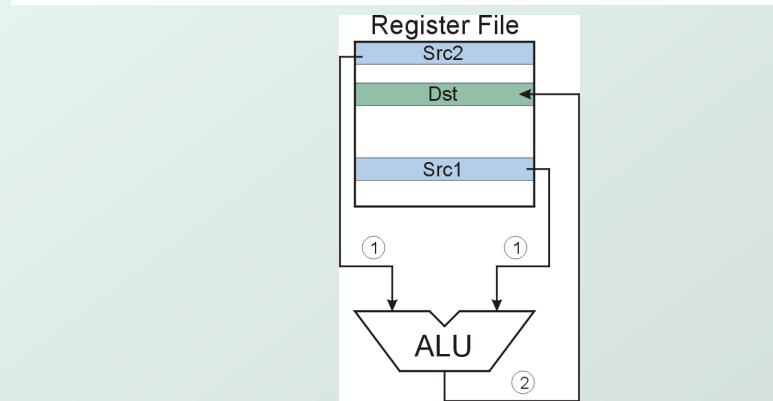
NOT	1 0 0 1	Dst	Src	1 1 1 1 1 1
-----	---------	-----	-----	-------------



ADD/AND (Register)

ADD	0 0 0 1	Dst	Src1	0 0 0	Src2
-----	---------	-----	------	-------	------

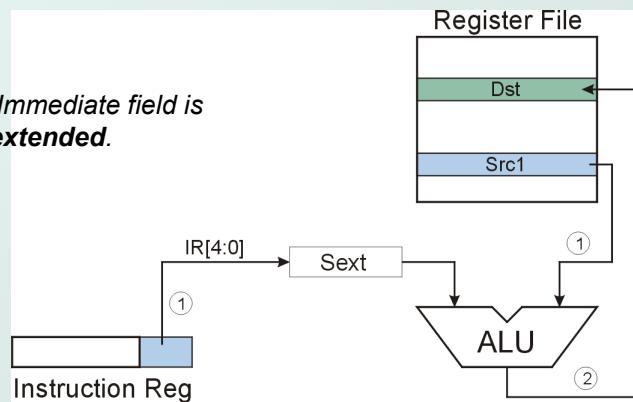
AND	0 1 0 1	Dst	Src1	0 0 0	Src2
-----	---------	-----	------	-------	------



ADD/AND (Immediate) *this one means "immediate mode"*

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	ADD	0 0 0 1	Dst	Src1	1	Imm5
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	AND	0 1 0 1	Dst	Src1	1	Imm5

Note: Immediate field is sign-extended.



Using Operate Instructions

- With only ADD, AND, NOT...

- How do we subtract?
- How do we OR?
- How do we copy from one register to another?
- How do we initialize a register to zero?

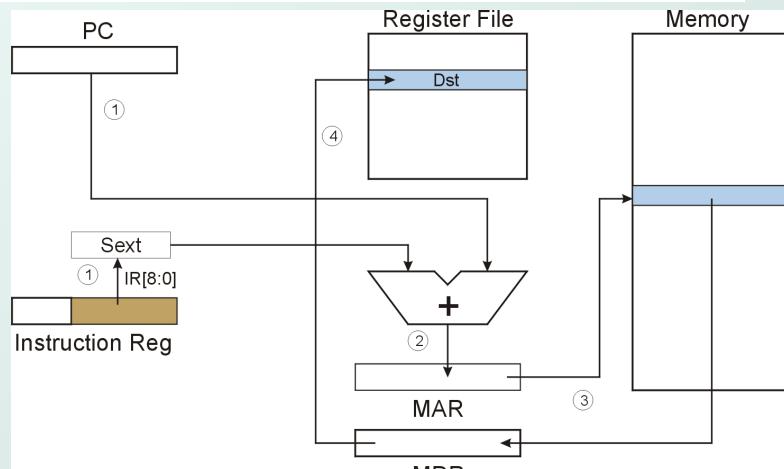
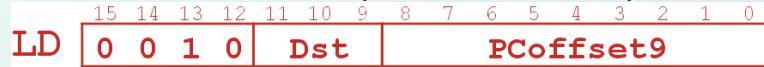
Data Movement Instructions

- 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
 - **LEA**: immediate mode
 - *does not access memory*

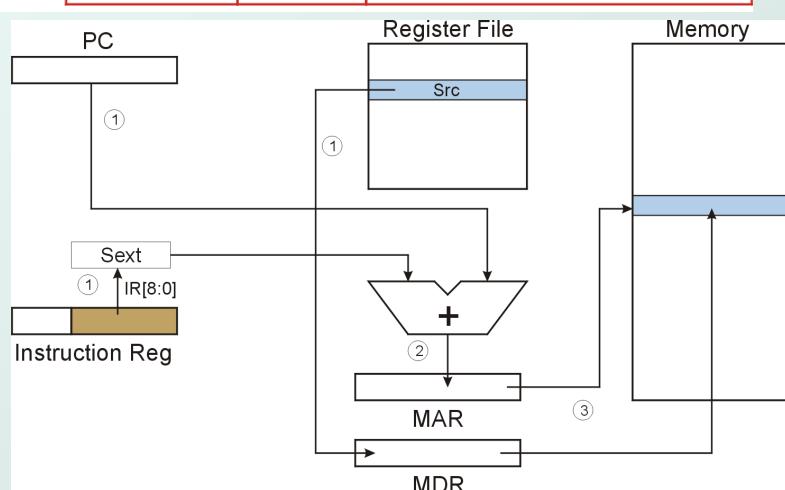
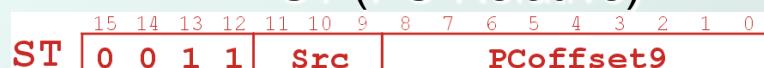
PC-Relative Addressing Mode

- 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:**
 - Use the 9 bits as a signed offset from the current PC.
- 9 bits: $-256 \leq \text{offset} \leq +255$
- Can form address such that: $\text{PC} - 256 \leq \text{X} \leq \text{PC} + 255$
 - Remember that PC is incremented as part of the FETCH phase;
 - This is done before the EVALUATE ADDRESS stage.

LD (PC-Relative)



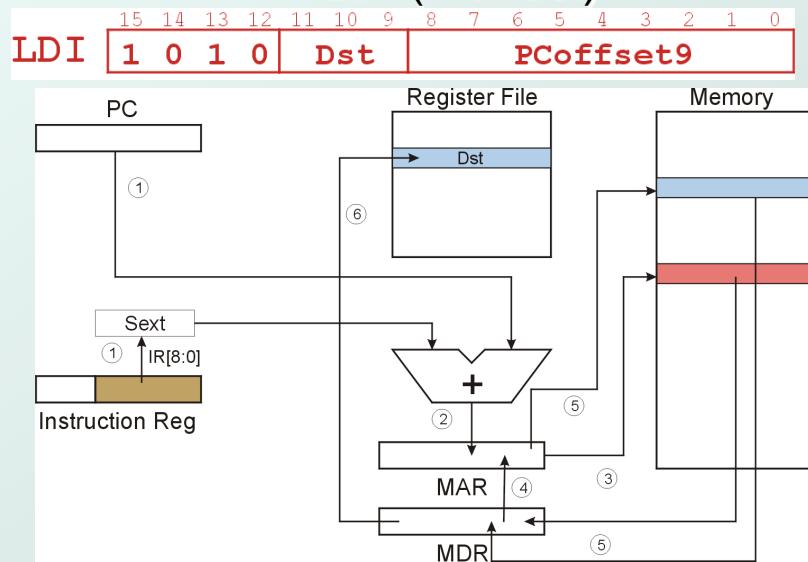
ST (PC-Relative)

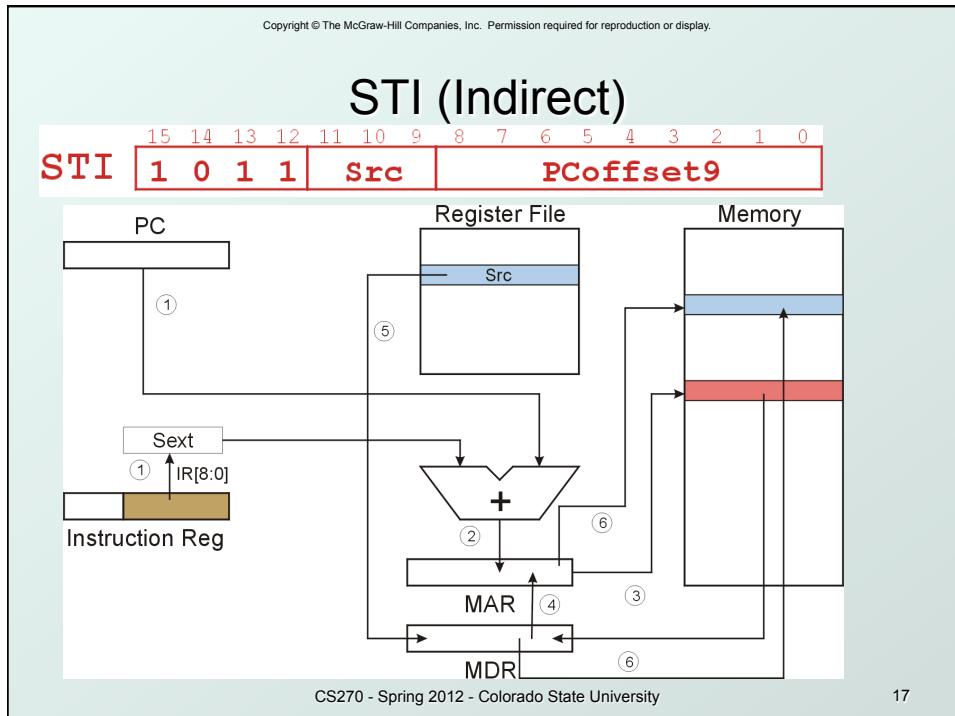


Indirect Addressing Mode

- With PC-relative mode, can only address data within 256 words of the instruction.
 - What about the rest of memory?
- Solution #1:**
 - Read address from memory location, then load/store to that address.
- First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.

LDI (Indirect)

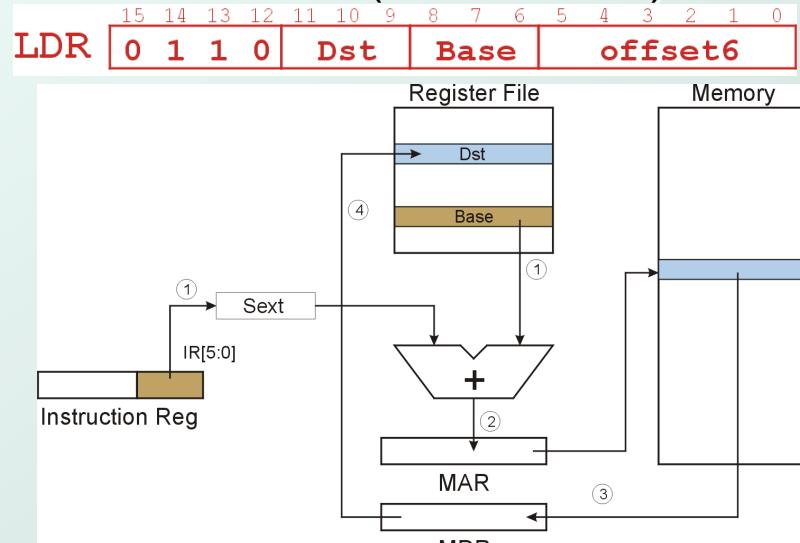




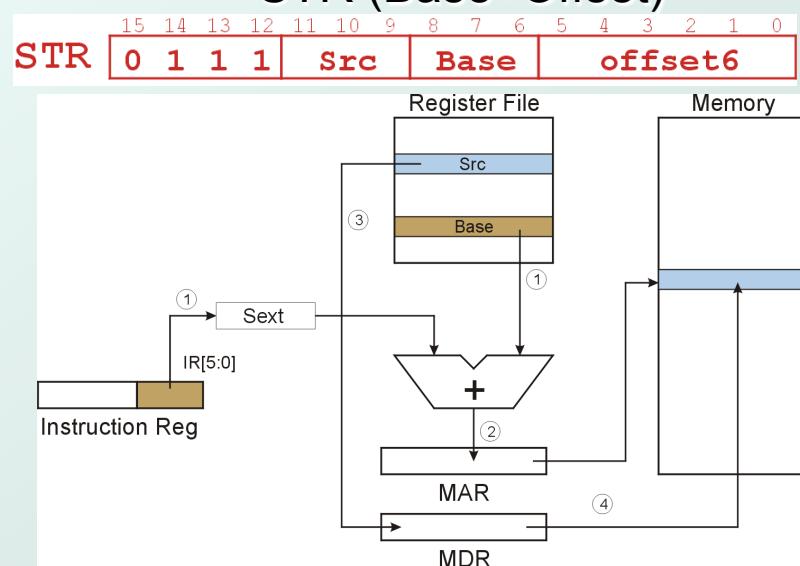
Base + Offset Addressing Mode

- With PC-relative mode, can only address data within 256 words of the instruction.
 - What about the rest of memory?
- Solution #2:**
 - Use a register to generate a full 16-bit address.
- 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.

LDR (Base+Offset)



STR (Base+Offset)

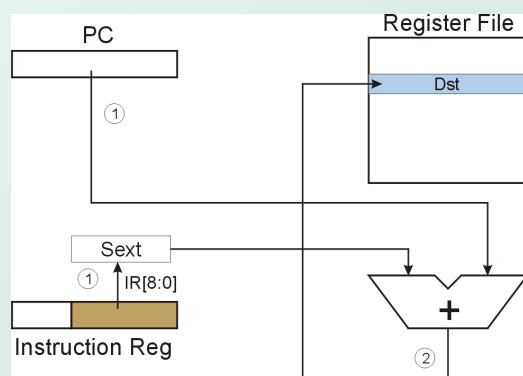


Load Effective Address

- Computes address like PC-relative (PC plus signed offset) and **stores the result into a register**.

Note: The address is stored in the register, not the contents of the memory location.

LEA (Immediate)



Example

Address	Instruction	Comments
x30F6	111000111111101	$R1 \leftarrow PC - 3 = x30F4$
x30F7	0001010001101110	$R2 \leftarrow R1 + 14 = x3102$
x30F8	001101011111011	$M[PC - 5] \leftarrow R2$ $M[x30F4] \leftarrow x3102$
x30F9	0101010010100000	$R2 \leftarrow 0$
x30FA	0001010010100101	$R2 \leftarrow R2 + 5 = 5$
x30FB	0111010001001110	$M[R1+14] \leftarrow R2$ $M[x3102] \leftarrow 5$
x30FC	101001111110111 opcode	$R3 \leftarrow M[M[x30F4]]$ $R3 \leftarrow M[x3102]$ $R3 \leftarrow 5$

Register Transfer Notation/Level

- Used to describe the operational behavior of digital circuits
 - Cycle by cycle or at a more “macro” level
 - $R1 \leftarrow PC - 3 = x30F4$
 - $DstReg \leftarrow Value$, using other regs or memory and ops
- Also names which control signals are on (i.e., 1) during a cycle. By default signals not named are off
- Control signals are the critical elements, everything else can be inferred.
 - More of this in recitation

Control Instructions

- Used to alter the sequence of instructions (by changing the Program Counter)
- Conditional Branch**
 - branch is *taken* if a specified condition is true
 - signed offset is added to PC to yield new PC
 - else, the branch is *not taken*
 - PC is not changed, points to the next instruction
- Unconditional Branch (or Jump)**
 - always changes the PC
- TRAP**
 - changes PC to the address of an OS “service routine”
 - routine will return control to the next instruction (after the TRAP)

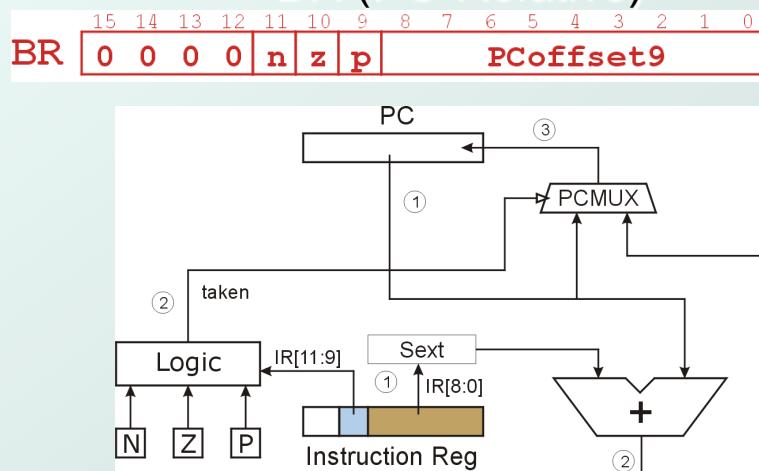
Condition Codes

- LC-3 has three **condition code** registers:
 - N** -- negative
 - Z** -- zero
 - P** -- positive (greater than zero)
- Set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)
- 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.
- If the set bit is specified, the branch is taken.
 - PC-relative addressing: **target address** is made by adding signed offset (IR [8:0]) to current PC.
 - Note: PC has already been incremented by FETCH stage.
 - Note: Target must be within 256 words of BR instruction.
- If the branch is not taken, the next sequential instruction is executed.

BR (PC-Relative)

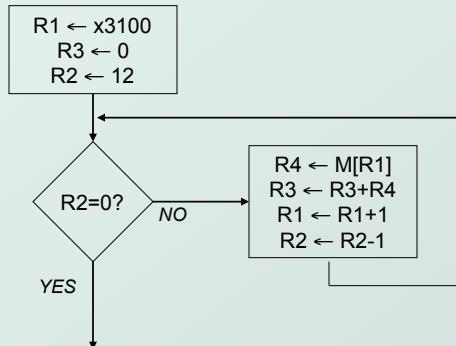


What happens if bits [11:9] are all zero? All one?

Using Branch Instructions

- Compute sum of 12 integers.

Numbers start at location x3100. Program starts at location x3000.



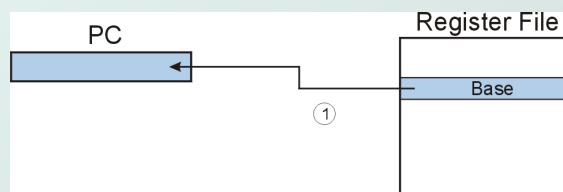
Sample Program

Address	Instruction	Comments
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	$R1 \leftarrow x3100$ (PC+0xFF)
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 0 1 0 1 0 0 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 1 0 1	If Z, goto x300A (PC+5)
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	Load next value to R4
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 0 0 1	Add to R3
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	Increment R1 (pointer)
X3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	Decrement R2 (counter)
x3009	0 0 0 0 1 1 1 1 1 1 1 1 0 1 0	Goto x3004 (PC-6)

JMP (Register)

- Jump is an unconditional branch -- always taken.
 - Target address is the contents of a register.
 - Allows any target address.

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



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	0	0	0	trapvect8	0	0	0

- Calls a **service routine**, identified by 8-bit “trap vector.”

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

- When routine is done,
PC is set to the instruction following TRAP.
 - We'll talk about how this works later.

Another Example

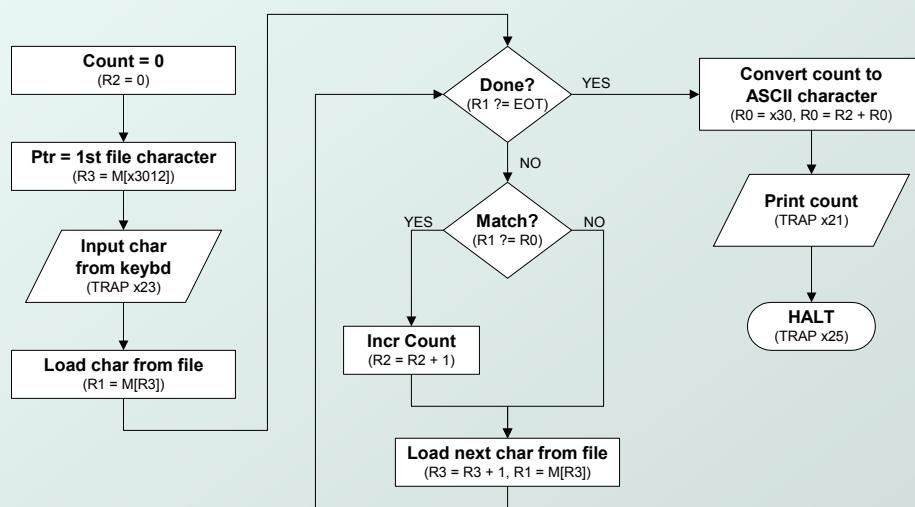
Count the occurrences of a character in a file

- 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 an ASCII value: **EOT (x04)**
- At the end, print the number of characters and halt
(assume there will be less 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

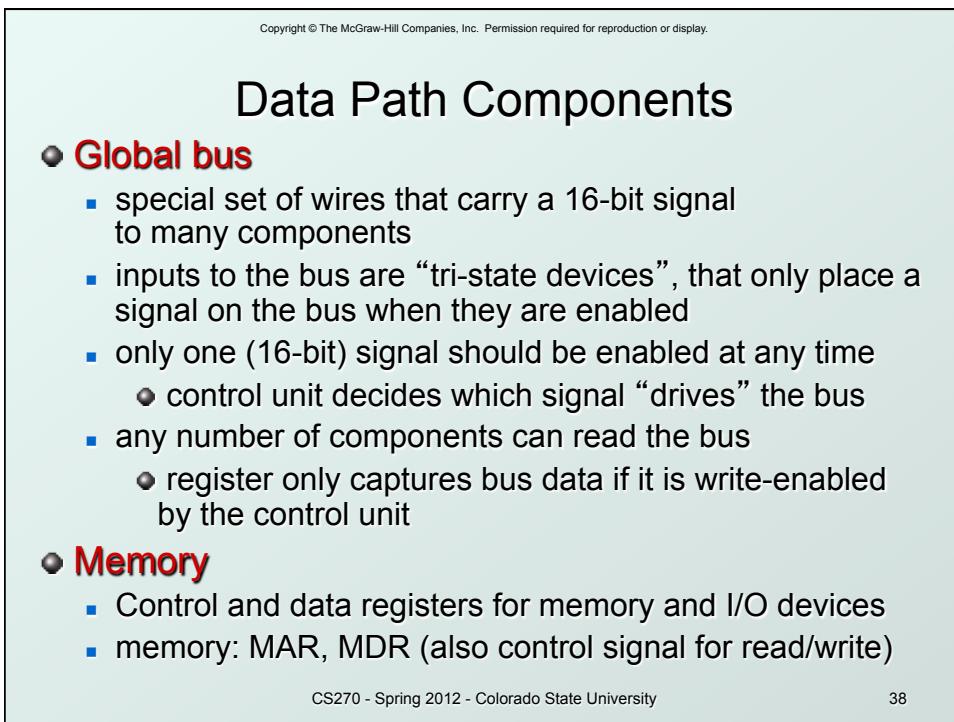
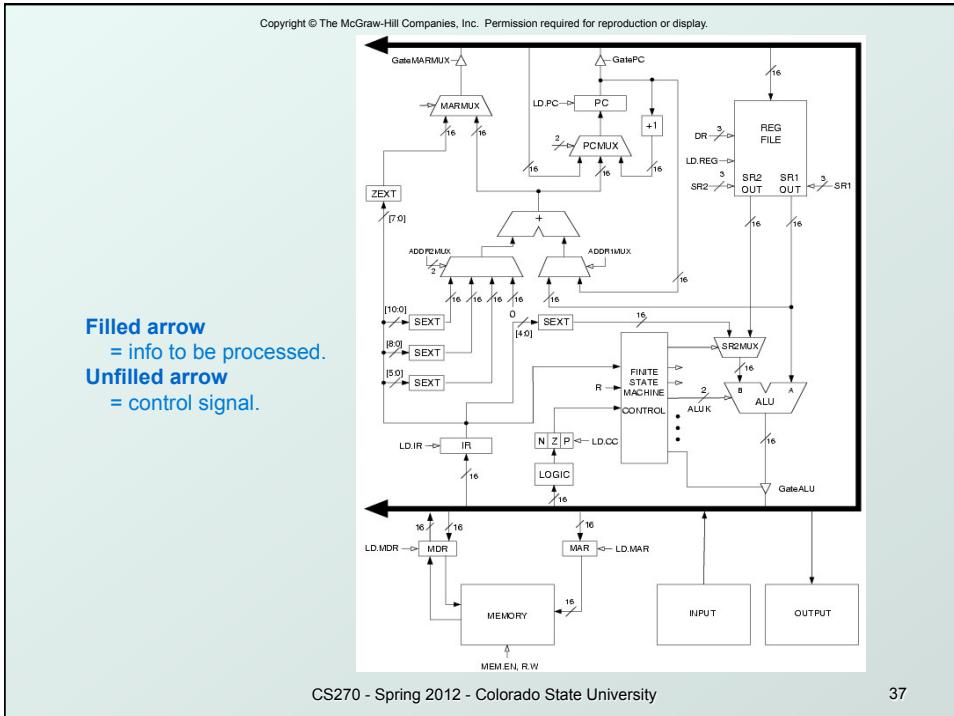


Program (1 of 2)

Address	Instruction	Comments
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	$R2 \leftarrow 0$ (counter)
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	$R3 \leftarrow M[x3102]$ (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	$R1 \leftarrow M[R3]$
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	$R4 \leftarrow R1 - 4$ (EOT)
x3005	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0	If Z, goto x300E
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	$R1 \leftarrow \text{NOT } R1$
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	$R1 \leftarrow R1 + 1$
X3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	$R1 \leftarrow R1 + R0$
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	If N or P, goto x300B

Program (2 of 2)

Address	Instruction	Comments
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	$R2 \leftarrow R2 + 1$
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	$R3 \leftarrow R3 + 1$
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	$R1 \leftarrow M[R3]$
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	Goto x3004
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0	$R0 \leftarrow M[x3013]$
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0	$R0 \leftarrow R0 + R2$
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')



Data Path Components

- **ALU**

- Accepts inputs from register file and from sign-extended bits from IR (immediate field).
- Output goes to bus.
 - used by condition code logic, register file, memory

- **Register File**

- Two read addresses (SR1, SR2), one write address (DR)
- Input from bus
 - result of ALU operation or memory read
- Two 16-bit outputs
 - used by ALU, PC, memory address
 - data for store instructions passes through ALU

Data Path Components

- **PC and PCMUX**

- Three inputs to PC, controlled by PCMUX
 - 1.PC+1 – FETCH stage
 - 2.Address adder – BR, JMP
 - 3.bus – TRAP (discussed later)

- **MAR and MARMUX**

- Two inputs to MAR, controlled by MARMUX
 - 1.Address adder – LD/ST, LDR/STR
 - 2.Zero-extended IR[7:0] -- TRAP (discussed later)

Data Path Components

• **Condition Code Logic**

- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them (LD.CC)
 - only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)

• **Control Unit – Finite State Machine**

- On each machine cycle, changes control signals for next phase of instruction processing
 - who drives the bus? (GatePC, GateALU, ...)
 - which registers are write enabled? (LD.IR, LD.REG, ...)
 - which operation should ALU perform? (ALUK)
- Logic includes decoder for opcode, etc.