Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 23, 7May2013:

Revision

James Goodman

Department of Computer Science

Credits: Some slides prepared by Gregory T. Byrd, North Carolina State University

To Be Posted Shortly

- Model answers for assignments
- Marks for Assignment 1
- Slides from my Lectures(bundled)

Assignment 2

- Due yesterday
- No late penalties
- No extensions
- Dropbox will stop accepting submissions after midnight tomorrow (before Thursday lecture)

WHAT YOU NEED TO KNOW FOR THE TEST

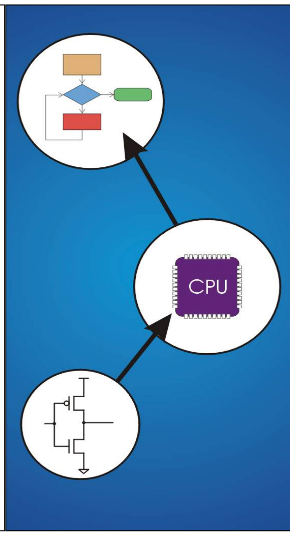
(in-class, Tuesday 14th May)

What You *Don't* Need to Know

- Real processors: MIPS & Alpha
- ASCII Table
- LC-3 instruction format
- Trap Codes, etc.

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

	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			



Chapter 1

Welcome Aboard

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 1, 5Mar13:

Introduction

James Goodman

Department of Computer Science

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 2, 7Mar13:

Introduction

James Goodman

Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

Two Recurring Themes

Abstraction

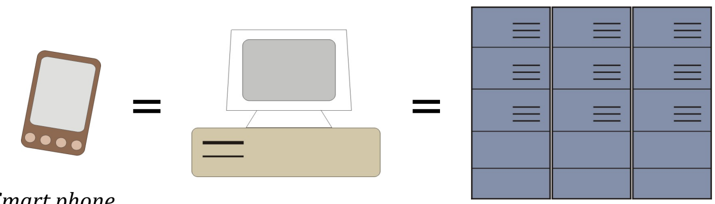
- Productivity enhancer – don't need to worry about details...
  Can drive a car without knowing how the internal combustion engine works.
- ...until something goes wrong!
  Where's the dipstick? What's a spark plug?
- Important to understand the components and how they work together.

Hardware vs. Software

- It's not either/or – both are components of a computer system.
- Even if you specialize in one, it is important to understand capabilities and limitations of both.

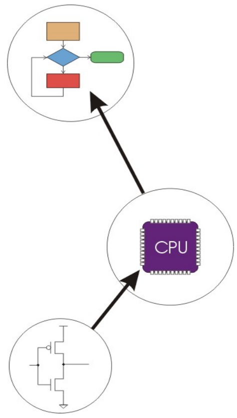
Big Idea #1: Universal Computing Device

All computers, given enough time and memory, are capable of computing exactly the same things.



Smart phone = *Desktop* = *Supercomputer*

Big Idea #2: Transformations Between Layers



Problems

Algorithms

Language

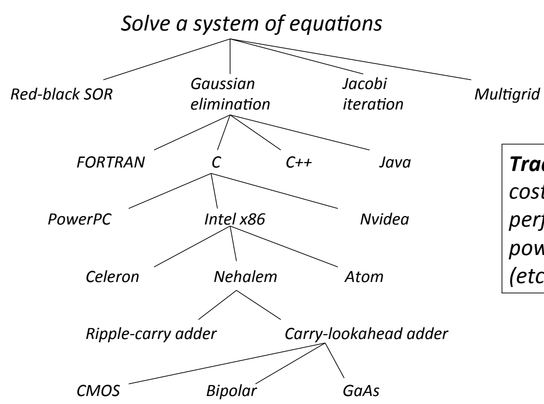
Instruction Set Architecture

Microarchitecture

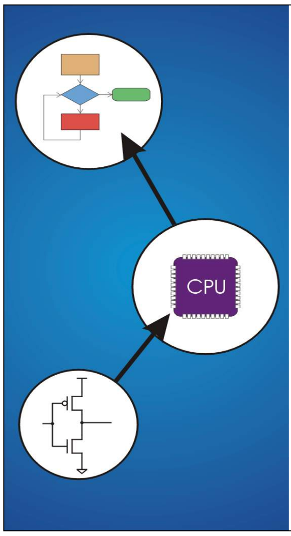
Circuits

Devices

Many Choices at Each Level



Tradeoffs:
cost
performance
power
(etc.)



Chapter 2
Bits, Data Types, and Operations

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 4, 12Mar13:

Arithmetic & Other Operations

James Goodman

Department of Computer Science

Credits: Adapted from slides prepared by Gregory T. Byrd, North Carolina State University

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 3, 8Mar13:

Representation & Arithmetic

James Goodman

Department of Computer Science

Credits: Adapted from slides prepared by Gregory T. Byrd, North Carolina State University

Unsigned Integers

Non-positional notation

- could represent a number ("5") with a string of ones ("11111")
- problems?

Weighted positional notation

- like decimal numbers: "329"
- "3" is worth 300, because of its position, while "9" is only worth 9

329 — 10^2 10^1 10^0 — $3x100 + 2x10 + 9x1 = 329$

most significant ... *101* ... *least significant* — 2^2 2^1 2^0 — $1x4 + 0x2 + 1x1 = 5$

Unsigned Integers (cont.)

An n-bit unsigned integer represents any of 2^n (integer) values:
from 0 to 2^n-1.

2^2	2^1	2^0	Value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

Base-2 addition – just like base-10!
- add from right to left, propagating carry

carry

```
  10010        10010        1111
+  1001      +  1011      +    1
  11011        11101       10000

               10111
             +   111
```

Subtraction, multiplication, division,…

Signed Integers

With n bits, we can distinguish 2^n unique values
- assign about half to positive integers (1 through 2^{n-1})
  and about half to negative (-2^{n-1} through -1)
- that leaves two values: one for 0, and one extra

Positive integers
- just like unsigned, but zero in *most significant* (MS) bit
  00101 = 5

Negative integers
- Sign-Magnitude (or Signed-Magnitude) – set MS bit to show negative, other bits are the same as unsigned
  10101 = -5
- One's complement – flip every bit to represent negative
  11010 = -5
- In either case, MS bit indicates sign: 0=positive, 1=negative

Two's Complement

Problems with sign-magnitude and 1's complement
- two representations of zero (+0 and –0)
- arithmetic circuits are complex
  - How to add two sign-magnitude numbers?
    - e.g., try 2 + (-3)
  - How to add two one's complement numbers?
    - e.g., try 4 + (-3)

Two's complement representation developed to make circuits easy for arithmetic.
- for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

```
  00101  (5)        01001   (9)
+ 11011  (-5)     + 10111   (-9)
  00000  (0)     (1)00000   (0)
```


Two's Complement Representation

If number is positive or zero,
- normal binary representation, zeroes in upper bit(s)

If number is negative,
- start with positive number
- flip every bit (i.e., take the one's complement)
- then add one

```
  00101 (5)               01001  (9)
  11010 (1's comp)        10110  (1's comp)
+     1                 +     1
  11011 (-5)              10111  (-9)
```


Two's Complement Signed Integers

MS bit is sign bit – it has weight -2^{n-1}.

Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.
- The most negative number (-2^{n-1}) has no positive counterpart.

-2^3	2^2	2^1	2^0		-2^3	2^2	2^1	2^0	
0	0	0	0	0	1	0	0	0	-8
0	0	0	1	1	1	0	0	1	-7
0	0	1	0	2	1	0	1	0	-6
0	0	1	1	3	1	0	1	1	-5
0	1	0	0	4	1	1	0	0	-4
0	1	0	1	5	1	1	0	1	-3
0	1	1	0	6	1	1	1	0	-2
0	1	1	1	7	1	1	1	1	-1

"Biased" Representation of Signed Integers

All integers (positive & negative) are represented as an unsigned integer supplemented with a "bias" to be subtracted out.

Range of an n-bit number: (0 - bias) through (2^n-1 - bias).

Bias 7:

2^3	2^2	2^1	2^0	Bias-7	2^3	2^2	2^1	2^0	Bias-7
0	0	0	0	-7	1	0	0	0	1
0	0	0	1	-6	1	0	0	1	2
0	0	1	0	-5	1	0	1	0	3
0	0	1	1	-4	1	0	1	1	4
0	1	0	0	-3	1	1	0	0	5
0	1	0	1	-2	1	1	0	1	6
0	1	1	0	-1	1	1	1	0	7
0	1	1	1	0	1	1	1	1	8

Converting Binary (2's C) to Decimal

1. If leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

Assuming 8-bit 2's complement numbers.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Converting Decimal to Binary (2's C)

First Method: *Division*

1. Find magnitude of decimal number. (Always positive.)
2. Divide by two – remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; if original number was negative, take two's complement.

$X = 104_{ten}$

104/2 =	52 r0	bit 0
52/2 =	26 r0	bit 1
26/2 =	13 r0	bit 2
13/2 =	6 r1	bit 3
6/2 =	3 r0	bit 4
3/2 =	1 r1	bit 5
1/2 =	0 r1	bit 6

$X = 01101000_{two}$

Interesting Properties of ASCII Code

What is relationship between a decimal digit ('0', '1', …) and its ASCII code?

What is the difference between an upper-case letter ('A', 'B', …) and its lower-case equivalent ('a', 'b', …)?

Given two ASCII characters, how do we tell which comes first in alphabetical order?

Is 128 characters enough? (http://www.unicode.org/)

No new operations -- integer arithmetic and logic.

Converting Decimal to Binary (2's C)

Second Method: *Subtract Powers of Two*

1. Find magnitude of decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit; if original was negative, take two's complement.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$X = 104_{ten}$

104 - 64 =	40	bit 6
40 - 32 =	8	bit 5
8 - 8 =	0	bit 3

$X = 01101000_{two}$

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 5, 14Mar13:

Representation of Fractions & Floating Point Numbers

James Goodman

Department of Computer Science

Credits: Adapted from slides prepared by Gregory T. Byrd, North Carolina State University

Fractions: Fixed-Point

How can we represent fractions?

- Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."
- 2's comp addition and subtraction still work
  - only if binary points are aligned

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$

```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  00100111.011 (39.375)
```

No new operations -- same as integer arithmetic.

Significant Digits

Accuracy of measurement leads to notion of *Significant Digits*

- For most purposes, we don't need high precision
- Accuracy of calculations is generally limited by least precise numbers
- Can represent numbers with a few significant digits
  - $6.0221415 * 10^{23}$ Avogadro's Number (approximately)
  - 299,792,458 meters/sec -- Speed of Light (exactly!)
    - By definition, a meter is the distance light travels through a vacuum in exactly 1/299792458 seconds
  - 3.141592…
    - Computable to arbitrary accuracy, but
    - More digits probably won't improve result.

Scientific Notation

Conventional (decimal) notation:

$\pm$ mantissa x 10^{exponent}

$1 \le$ mantissa < 10

exponent is signed integer

Binary notation:

$\pm$ mantissa x 2^{exponent}

$1 \le$ mantissa < 2

exponent is signed integer

Floating Point Example

Single-precision IEEE floating point number:

1 01111110 10000000000000000000000

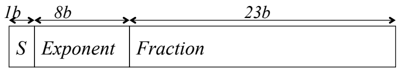
sign *exponent* *fraction*

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000… = 0.5 (decimal).

Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

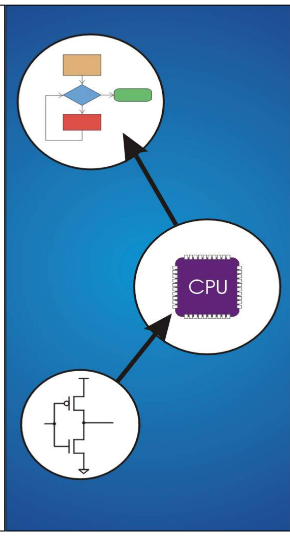
IEEE Floating-Point Representation

IEEE 754 Floating-Point Standard (32-bits):



$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127},\ \ 1 \le \text{exponent} \le 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126},\ \ \text{exponent} = 0$$



Chapter 3
Digital Logic Structures

Computer Science 210 s1
Computer Systems 1
2013 Semester 1
Lecture Notes

Lecture 6, 15Mar13:

Floating Point/ Digital Logic Structures

James Goodman

Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

Computer Science 210 s1
Computer Systems 1
2013 Semester 1
Lecture Notes

Lecture 7, 19Mar13:

Digital Logic Structures

James Goodman

Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

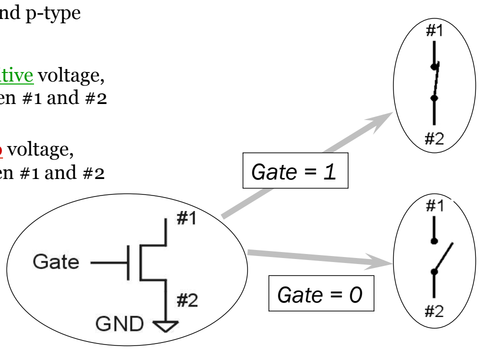
n-type MOS Transistor

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

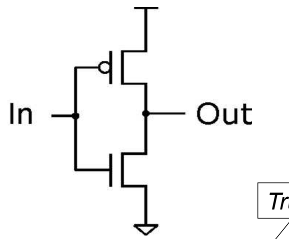
n-type

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)



Terminal #2 must be connected to GND (0V).

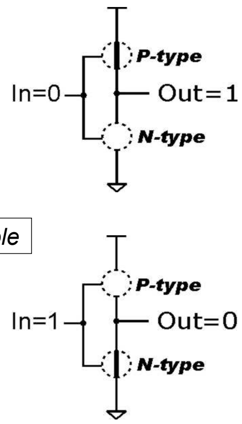
Inverter (NOT Gate)



In	Out
0 V	2.9 V
2.9 V	0 V

Truth table

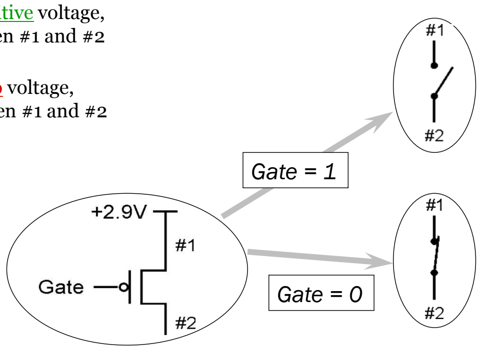
In	Out
0	1
1	0



p-type MOS Transistor

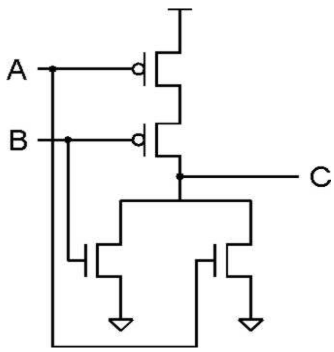
p-type is *complementary* to n-type

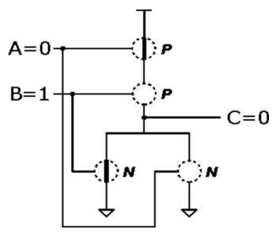
- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)



Terminal #1 must be connected to +2.9V.

NOR Gate (OR-NOT)

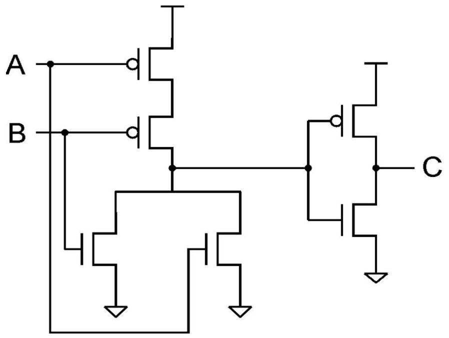




A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

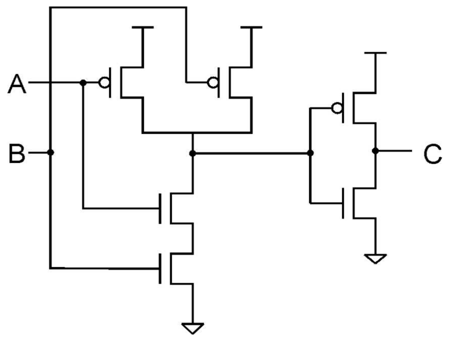
OR Gate



A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Add inverter to NOR.

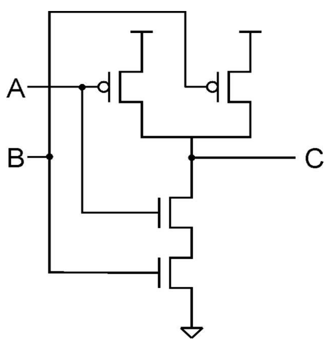
AND Gate

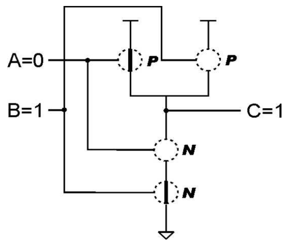


A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Add inverter to NAND.

NAND Gate (AND-NOT)

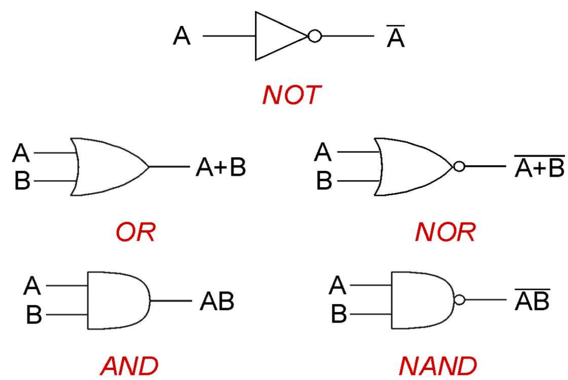




A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Note: Parallel structure on top, serial on bottom.

Basic Logic Gates



Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 8, 21Mar13:

Sequential Logic & Finite State Machines

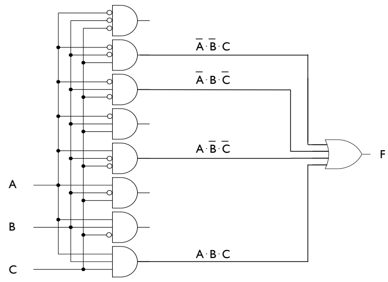
James Goodman

Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

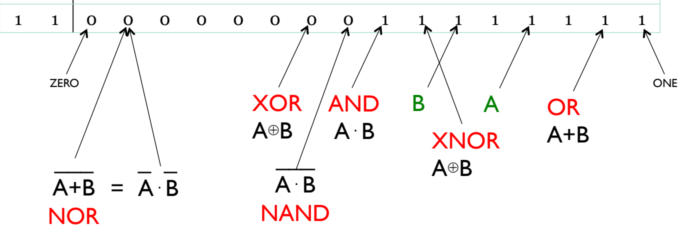
Creating a Function from a Truth Table

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



Two Variables: 16 Unique Functions

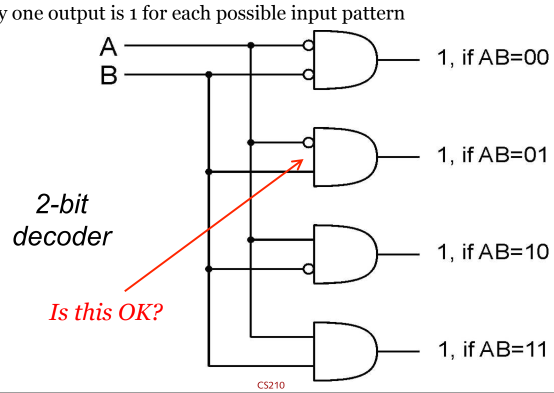
A	B	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1



Decoder

n inputs, 2^n outputs

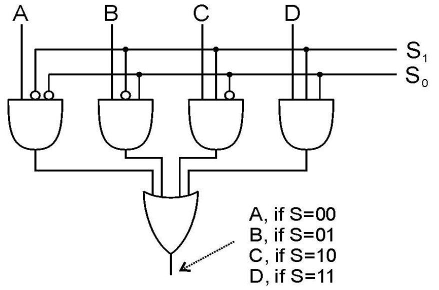
- exactly one output is 1 for each possible input pattern

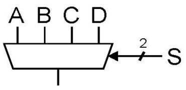


Multiplexer (MUX)

n-bit selector and 2^n inputs, one output

- output equals one of the inputs, depending on selector





4-to-1 MUX

Combinational vs. Sequential

Combinational Circuit

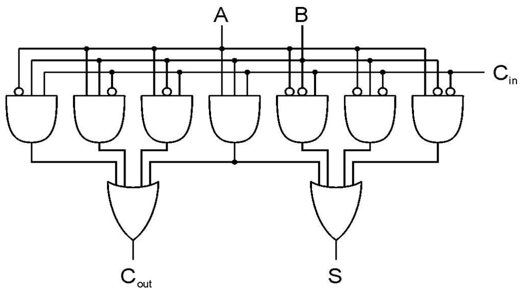
- always gives the same output for a given set of inputs
  - ex: adder always generates sum and carry, regardless of previous inputs

Sequential Circuit

- stores information
- output depends on stored information (state) plus input
  - so a given input might produce different outputs, depending on the stored information
- *example:* ticket counter
  - advances when you push the button
  - output depends on previous state
- useful for building "memory" elements and "state machines"

Full Adder

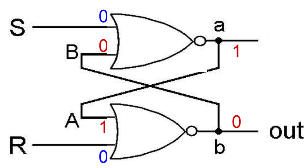
Add two bits and carry-in,
produce one-bit sum and carry-out.

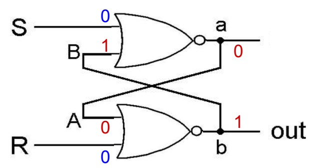


A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

R-S Latch: Simple Storage Element (NOR)

R is used to "reset" or "clear" the element – set output to zero.
S is used to "set" the element – set output to one.





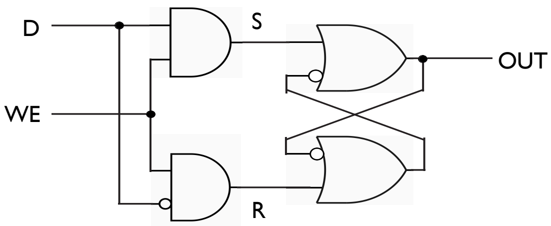
If both R and S are zero, out could be *either* zero or one.

- "quiescent" state – holds its previous value
- note: if a is 1, b is 0, and vice versa

Gated D-Latch

Two inputs: D (data) and WE (write enable)

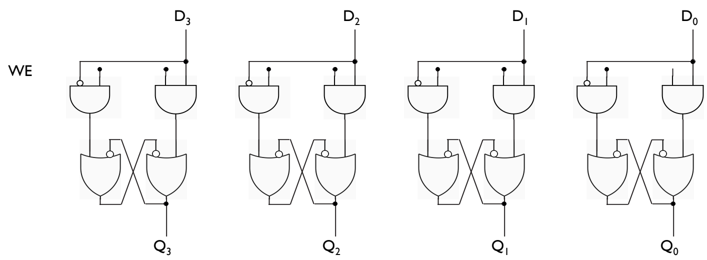
- when WE = 1, latch is set to value of D
  - S = D, R = NOT(D)
- when WE = 0, latch holds previous value
  - S = R = 0



Register

A register stores a multi-bit value.

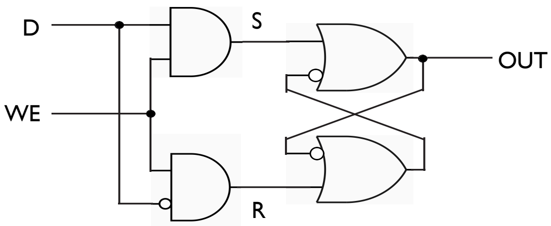
- We use a collection of D-latches, all controlled by a common WE.
- When WE=1, n-bit value D is written to register.



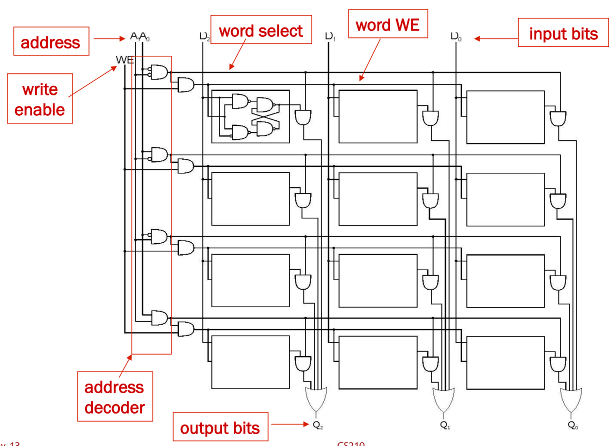
Gated D-Latch

Two inputs: D (data) and WE (write enable)

- when WE = 1, latch is set to value of D
  - S = D, R = NOT(D)
- when WE = 0, latch holds previous value
  - S = R = 0



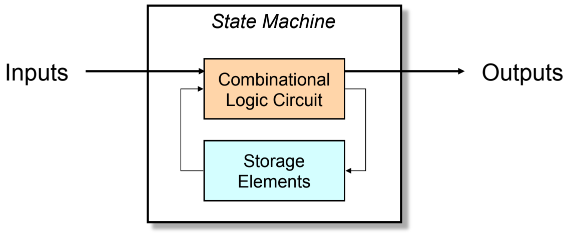
2^2 x 3 Memory



State Machine

Another type of sequential circuit

- Combines combinational logic with storage
- "Remembers" state, and changes output (and state) based on inputs and current state



State of Sequential Lock

Our lock example has four different states, labelled A-D:

A: The lock is not open, and no relevant operations have been performed.

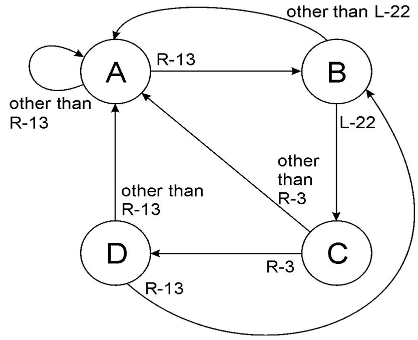
B: The lock is not open, and the user has completed the R-13 operation.

C: The lock is not open, and the user has completed R-13, followed by L-22.

D: The lock is open.

State Diagram

Shows states and actions that cause a transition between states.



Finite State Machine

A description of a system with the following components:

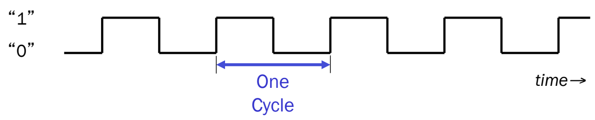
1. A finite number of states
2. A finite number of external inputs
3. A finite number of external outputs
4. An explicit specification of all state transitions
5. An explicit specification of what determines each external output value

Often described by a state diagram.

- Inputs trigger state transitions.
- Outputs are associated with each state (or with each transition).

The Clock

Frequently, a clock circuit triggers transition from one state to the next.

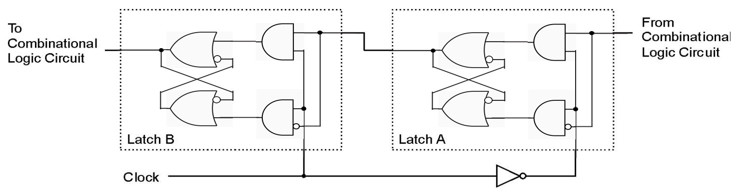


At the beginning of each clock cycle, state machine makes a transition, based on the current state and the external inputs.

- **Not always required. In lock example, the input itself triggers a transition.**

Storage: Master-Slave Flipflop

A pair of gated D-latches, to isolate *next* state from *current* state.



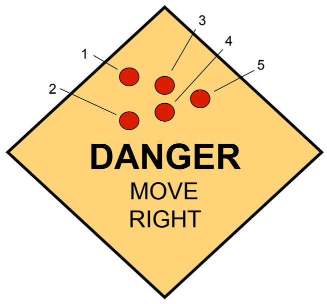
During 1st phase (clock=1), previously-computed state becomes *current* state and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

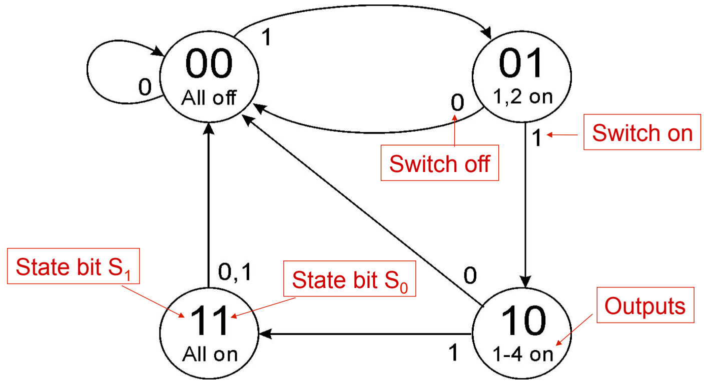
Complete Example

A blinking traffic sign

- No lights on
- 1 & 2 on
- 1, 2, 3, & 4 on
- 1, 2, 3, 4, & 5 on
- (repeat as long as switch is turned on)



Traffic Sign State Diagram



Transition on each clock cycle.

Traffic Sign Truth Tables

Outputs
(depend only on state: S_1S_0)

Lights 1 and 2 (Z)
Lights 3 and 4 (Y)
Light 5 (X)

S_1	S_0	Z	Y	X
0	0	0	0	0
0	1	1	0	0
1	0	1	1	0
1	1	1	1	1

Next State: $S_1'S_0'$
(depend on state and input)

Switch (On)

On	S_1	S_0	S_1'	S_0'
0	X	X	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

Whenever On=0, next state is 00.

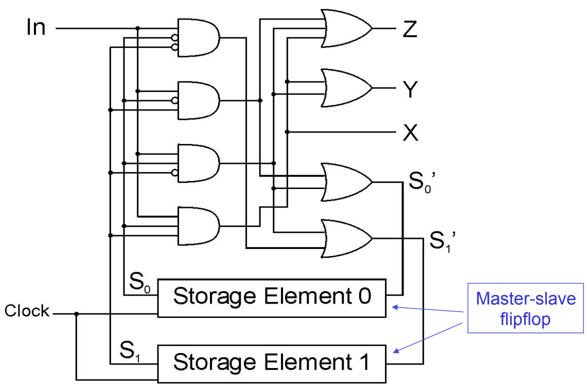
Computer Science 210 s1c
Computer Systems 1
2013 Semester 1
Lecture Notes

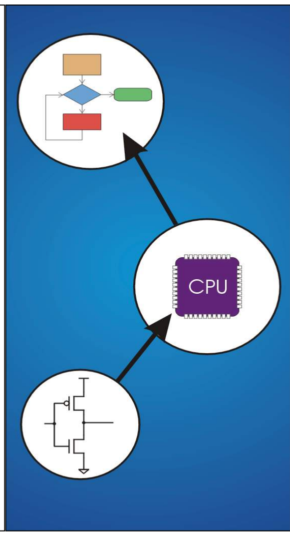
Lecture 9, 22Mar13:
Finite State Machines & the von Neumann Model

James Goodman
Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

Traffic Sign Logic





Chapter 4
The Von Neumann Model

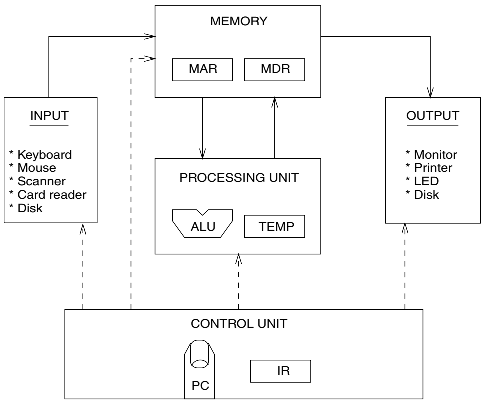
Computer Science 210 s1c
Computer Systems 1
2013 Semester 1
Lecture Notes

Lecture 10, 26Mar13:
The von Neumann Computer

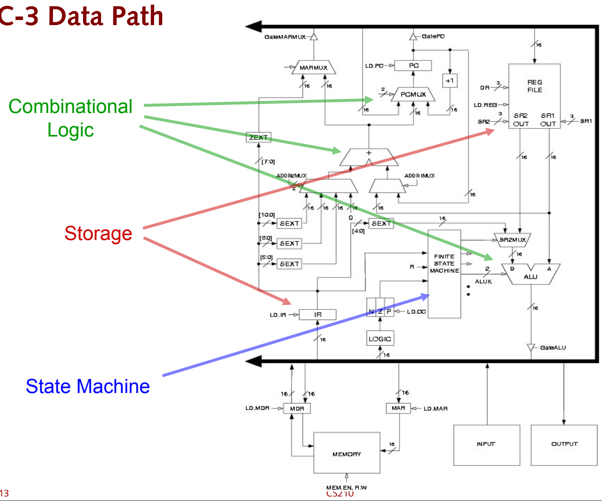
James Goodman
Department of Computer Science

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University

The Von Neumann Model



LC-3 Data Path



Memory

2^k x m array of stored bits

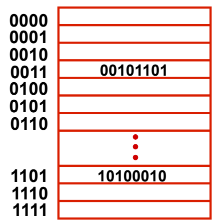
Address
- unique (k-bit) identifier of location

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

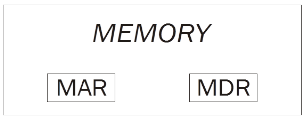


Interface to Memory

How does processing unit get data to/from memory?

MAR: Memory Address Register

MDR: Memory Data Register



To LOAD a location (A):

1. Write the address (A) into the MAR.
2. Send a "read" signal to the memory.
3. Read the data from MDR.

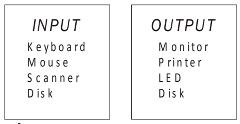
To STORE a value (X) to a location (A):

1. Write the data (X) to the MDR.
2. Write the address (A) into the MAR.
3. Send a "write" signal to the memory.

Input and Output

Devices for getting data into and out of computer memory

Each device has its own interface, usually a set of registers like the memory's MAR and MDR



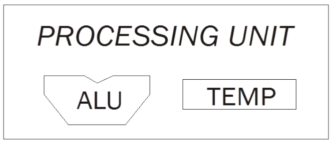
- LC-3 supports keyboard (input) and monitor (output)
- keyboard: data register (KBDR) and status register (KBSR)
- monitor: data register (DDR) and status register (DSR)

Some devices provide both input and output
- disk, network

Program that controls access to a device is usually called a *driver*.

Processing Unit

Functional Units
- ALU = Arithmetic and Logic Unit
- could have many functional units. some of them special-purpose (multiply, square root, ...)
- LC-3 performs ADD, AND, NOT

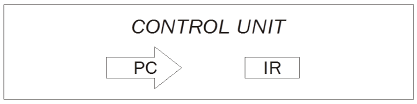


Registers
- Small, temporary storage
- Operands and results of functional units
- LC-3 has eight registers (R0, ..., R7), each 16 bits wide

Word Size
- number of bits normally processed by ALU in one instruction
- also width of registers
- LC-3 is 16 bits

Control Unit

Orchestrates execution of the program



Instruction Register (IR) contains the *current instruction*.

Program Counter (PC) contains the *address* of the next instruction to be executed.

Control unit:
- reads an instruction from memory
  - the instruction's address is in the PC
- interprets the instruction, generating signals that tell the other components what to do
  - an instruction may take many *machine cycles* to complete

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 11, 28Mar13:

The Instruction Cycle
Ch. 5: The LC-3 ISA

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

Instruction

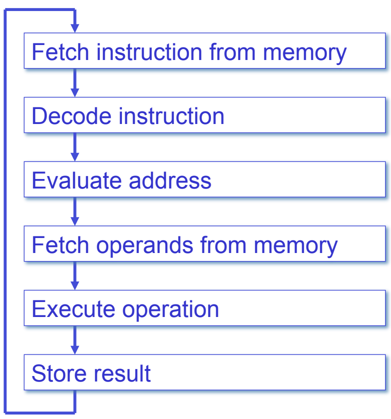
The instruction is the fundamental unit of work.

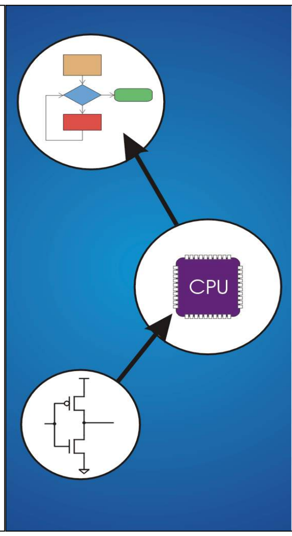
Specifies two things:
- *opcode*: operation to be performed
- *operands*: data/locations to be used for operation

An instruction is encoded as a sequence of bits. *(Just like data!)*
- Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
- Control unit interprets instruction: generates sequence of control signals to carry out operation.
- Operation is either executed completely, or not at all.

A computer's instructions and their formats is known as its ***Instruction Set Architecture (ISA)***.

Instruction Processing





Chapter 5
The LC-3

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 12, 4Apr13:

Ch. 5: The LC-3 ISA

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

Example: LC-3 LDR Instruction

Load instruction – reads data from memory

Base + offset mode:

- add offset to base register – result is memory address
- load from memory address into destination register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR				Dst			Base			Offset					
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 that memory location to R2."

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	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD				Dst			Src1			0	0	0	Src2		
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 JMP Instruction

Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.

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

"Load the contents of R3 into the PC."

Instruction Processing Summary

Instructions look just like data – it's all interpretation.

Three basic kinds of instructions:

- computational instructions (ADD, AND, ...)
- data movement instructions (LD, ST, ...)
- control instructions (JMP, BRnz, ...)

Six basic phases of instruction processing:

F → D → EA → OP → EX → S

- not all phases are needed by every instruction
- phases may take variable number of machine cycles

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 13, 5Apr13:

Ch. 5: The LC-3

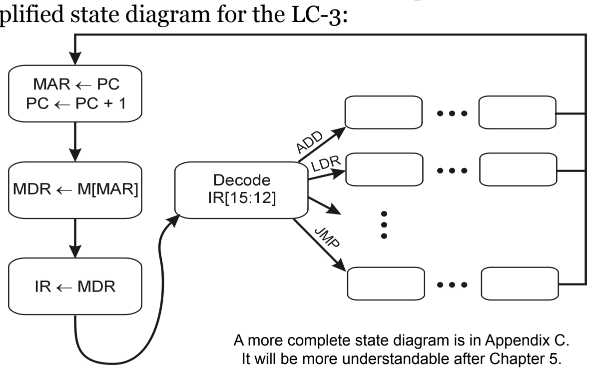
James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

Control Unit State Diagram

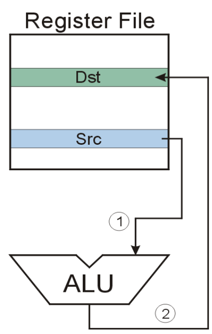
The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:



A more complete state diagram is in Appendix C.
It will be more understandable after Chapter 5.

NOT (Register)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOT	1	0	0	1	Dst			Src			1	1	1	1	1	1



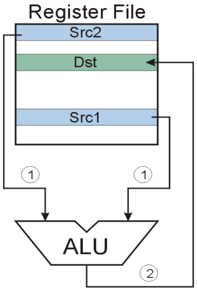
Note: Src and Dst could be the *<u>same</u>* *register.*

ADD/AND (Register)

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



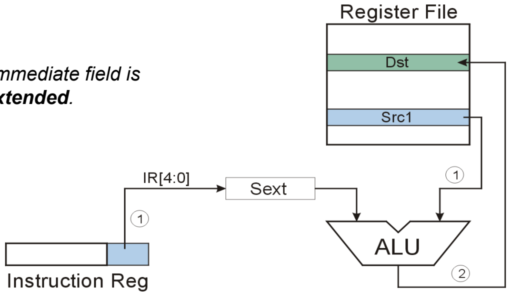
ADD/AND (Immediate)

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
AND	0	1	0	1	Dst	Src1	1	Imm5

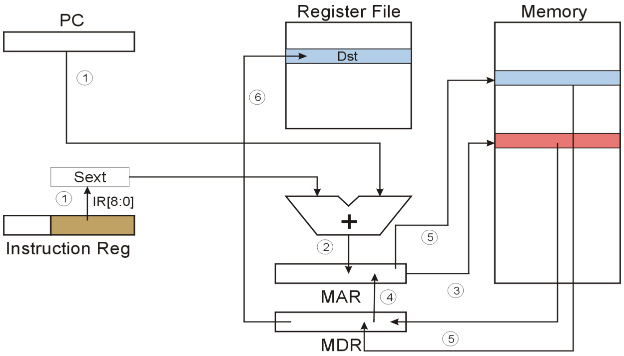
Note: Immediate field is ***sign-extended***.



LDI (Indirect)

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

LDI	1	0	1	0	Dst	PCoffset9



Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 14, 9Apr13:

The LC-3
Chapter 7: Assembly Language

James Goodman

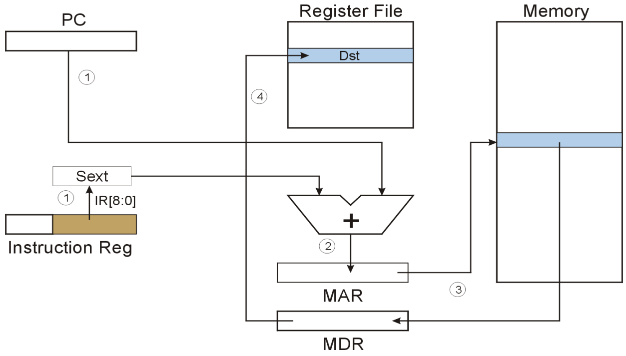
Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

LD (PC-Relative)

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

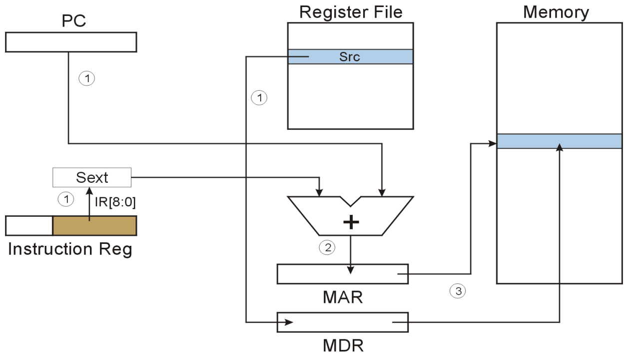
LD	0	0	1	0	Dst	PCoffset9



ST (PC-Relative)

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

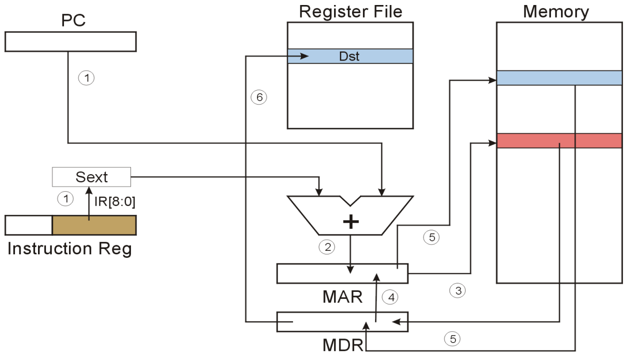
ST	0	0	1	1	Src	PCoffset9



LDI (Indirect)

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

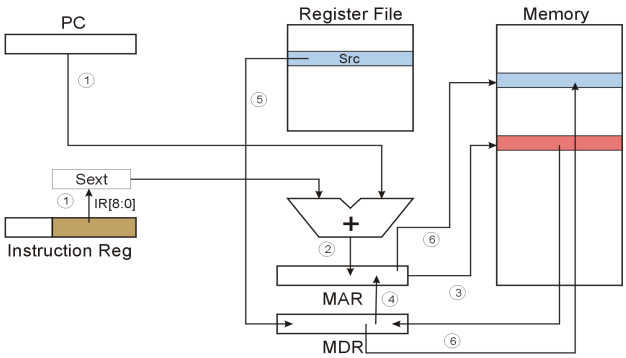
LDI	1	0	1	0	Dst	PCoffset9



STI (Indirect)

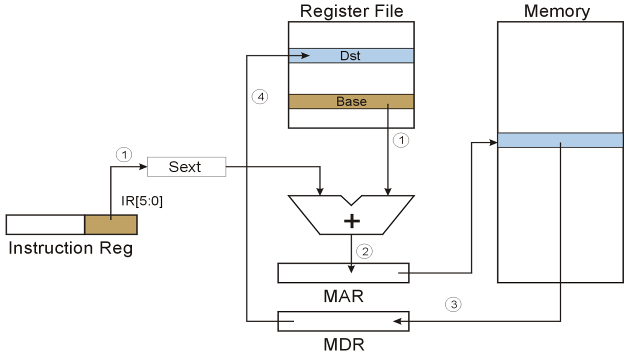
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

STI	1	0	1	1	Src	PCoffset9



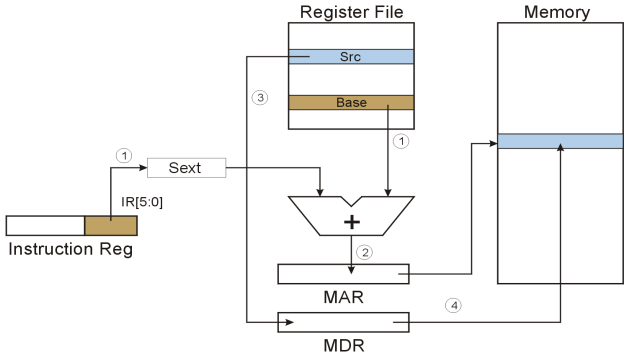
LDR (Base+Offset)

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
LDR	0 1 1 0	Dst	Base	offset6



STR (Base+Offset)

	15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
STR	0 1 1 1	Src	Base	offset6



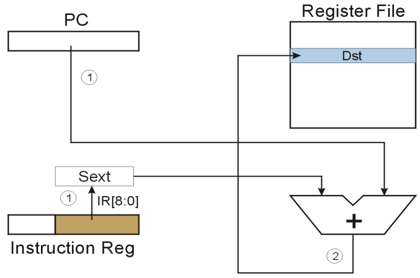
Load Effective Address

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)

	15 14 13 12	11 10 9	8 7 6 5 4 3 2 1 0
LEA	1 1 1 0	Dst	PCoffset9



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 sequential 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 TRAP)

Condition Codes

LC-3 has three condition code bits:
- 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

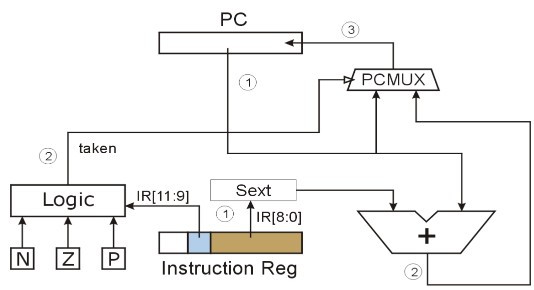
Branch specifies one or more condition codes.
If the set bit is specified, the branch is taken.

- 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)

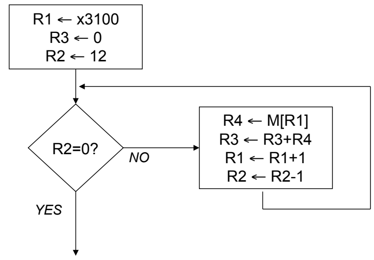
	15 14 13 12	11	10	9	8 7 6 5 4 3 2 1 0
BR	0 0 0 0	n	z	p	PCoffset9



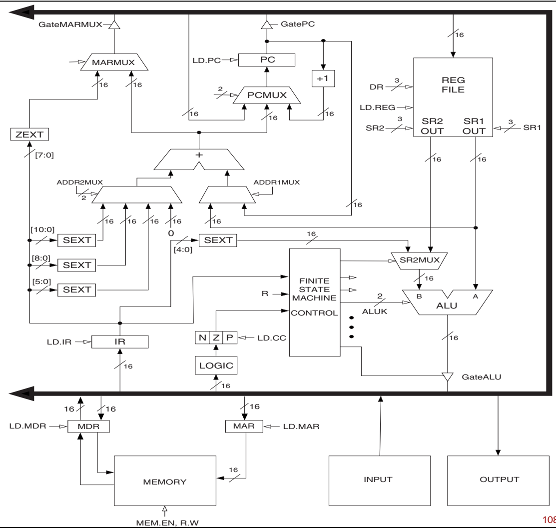
What happens if bits [11:9] are all zero? All one?

Using Branch Instructions

Compute sum of 12 integers.
Numbers start at location x3100. Program starts at location x3000.



LC-3 Data Path Revisited



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

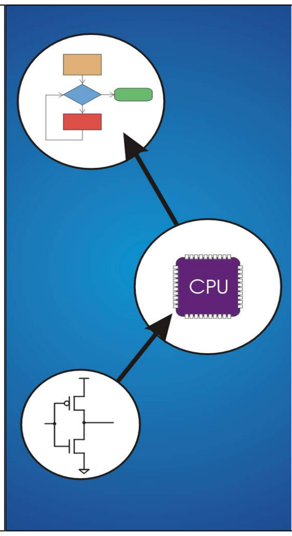
TRAP

	15	14	13	12	11	10	9	8	7 – 0
TRAP	1	1	1	1	0	0	0	0	trapvect8

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.)



Chapter 7
Assembly Language

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1
Lecture Notes

Lecture 15, 11Apr13:

Chapter 7: Assembly Language

James Goodman

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
        .ORIG  x3050
        LD     R1, SIX
        LD     R2, NUMBER
        AND    R3, R3, #0     ; Clear R3.  It will
                              ; contain the product.
; The inner loop
;
AGAIN   ADD    R3, R3, R2
        ADD    R1, R1, #-1    ; R1 keeps track of
        BRp    AGAIN          ; the iteration.
;
        HALT
;
NUMBER  .BLKW  1
SIX     .FILL  x0006
;
        .END
```


Human-Readable Machine Language

Computers like ones and zeros…
`0001110010000110`

Humans like symbols…
`ADD   R6,R2,R6   ; increment index reg.`

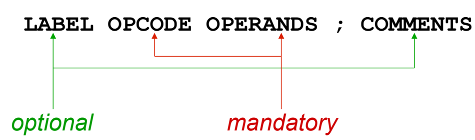
Assembler is a program that turns symbols into machine instructions.
- ISA-specific:
  close correspondence between symbols and instruction set
  - mnemonics for opcodes
  - labels for memory locations
- additional operations for allocating storage and initializing data

LC-3 Assembly Language Syntax

Each line of a program is one of the following:
- an instruction
- an assembler directive (or pseudo-op)
- a comment

Whitespace (between symbols) and case are ignored.
Comments (beginning with ";") are also ignored.

An instruction has the following format:



Opcodes and Operands

Opcodes
- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
  - ex: **ADD**, **AND**, **LD**, **LDR**, …

Operands
- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format
  - ex:

```
ADD R1,R1,R3
ADD R1,R1,#3
LD  R6,NUMBER
BRz LOOP
```


Labels and Comments

Label
- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line
  - ex:

```
LOOP  ADD R1,R1,#-1
      BRp LOOP
```

Comment
- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs
- tips for useful comments:
  - avoid restating the obvious, as "decrement R1"
  - provide additional insight, as in "accumulate product in R6"
  - use comments to separate pieces of program

Assembler Directives

Pseudo-operations
- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with a full stop

Opcode	*Operand*	*Meaning*
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate *n* words of storage
.FILL	n	allocate one word, initialize with value *n*
.STRINGZ	n-character string	allocate *n*+1 locations, initialize w/ characters and null terminator

Trap Codes

LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

Code	*Equivalent*	*Description*
HALT	**TRAP x25**	Halt execution and print message to console.
IN	**TRAP x23**	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	**TRAP x21**	Write one character (in R0[7:0]) to console.
GETC	**TRAP x20**	Read one character from keyboard. Character stored in R0[7:0].
PUTS	**TRAP x22**	Write null-terminated string to console. Address of string is in R0.

Style Guidelines

Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author's name, date, etc., and purpose of program.
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.
7. Each line must fit on the page -- no wraparound or truncations.
   - Long statements split in aesthetically pleasing manner.

The Assembly Process

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 16, 12Apr13:

The Assembly Process; Chapter 8: Input & Output

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

First Pass: Constructing the Symbol Table

1. Find the **.ORIG** statement, which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      – NOTE: If statement is **.BLKW** or **.STRINGZ**, increment LC by the number of words allocated.
3. Stop when **.END** statement is reached.

NOTE: A line that contains only a comment is considered an empty line.

Symbol Table Construction

Construct the symbol table for the program in Figure 7.1 .

Symbol	Address

Symbol Table Construction

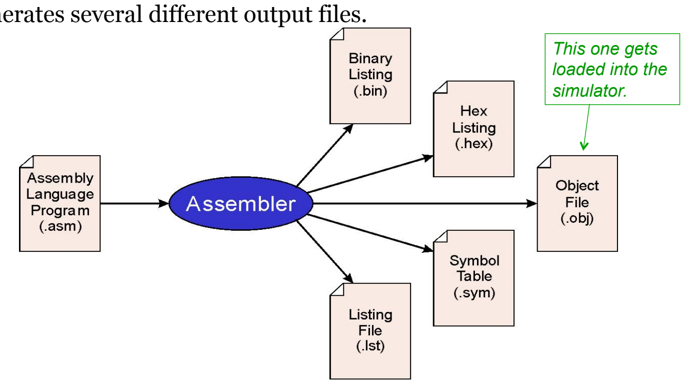
Construct the symbol table for the program in Figure 7.1 .

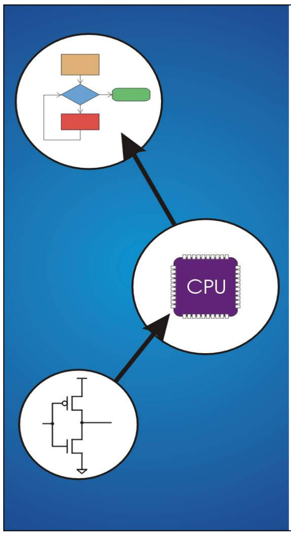
Symbol	Address
PTR	0x3013
TEST	0x3004
OUTPUT	0x300E
GETCHAR	0x300B
ASCII	0x3012


```
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
                .ORIG       x3000
        0x3000  AND         R2, R2, #0          ; R2 is counter, initially 0
        0x3001  LD          R3, PTR             ; R3 is pointer to characters
                GETC                            ; R0 gets character input
                LDR         R1, R3, #0          ; R1 gets first character
;
; Test character for end of file
;
TEST            ADD         R4, R1, #-4         ; Test for EOT (ASCII x04)
                BRz         OUTPUT              ; If done, prepare the output
;
; Test character for match.  If a match, increment count.
;
                NOT         R1, R1
                ADD         R1, R1, R0          ; If match, R1 = xFFFF
                NOT         R1, R1              ; If match, R1 = x0000
                BRnp        GETCHAR             ; If no match, do not increment
                ADD         R2, R2, #1
;
; Get next character from file.
;
GETCHAR         ADD         R3, R3, #1          ; Point to next character.
                LDR         R1, R3, #0          ; R1 gets next char to test
                BRnzp       TEST
;
; Output the count.
;
OUTPUT          LD          R0, ASCII           ; Load the ASCII template
                ADD         R0, R0, R2          ; Convert binary count to ASCII
                OUT                             ; ASCII code in R0 is displayed.
                HALT                            ; Halt machine
;
; Storage for pointer and ASCII template
;
ASCII           .FILL       x0030
PTR             .FILL       x4000
                .END
```


LC-3 Assembler

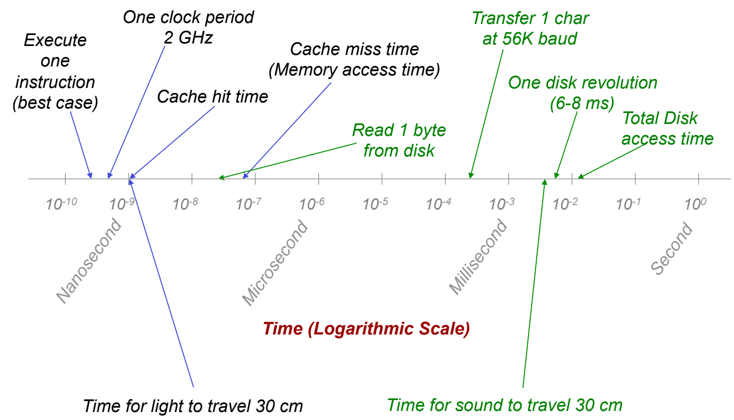
Using "assemble" (Unix) or LC3Edit (Windows), generates several different output files.





Chapter 8
I/O

Speed Line



Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 17, 16Apr13:

Chapter 8: Input & Output

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 18, 18Apr13:

Input & Output
Chap. 9: TRAP Routines

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

I/O: Connecting to the Outside World

Types of I/O devices characterized by:

- behavior: input, output, storage
  - input: keyboard, motion detector, network interface
  - output: monitor, printer, network interface
  - storage: disk, CD-ROM
- data rate: how fast can data be transferred?
  - Latency: how long to get the first byte
  - Bandwidth: rate that data is received

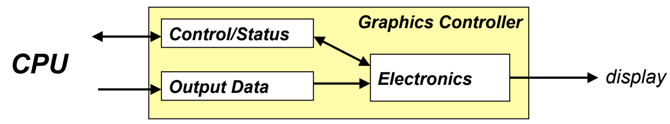
I/O Controller

Control/Status Registers

- CPU tells device what to do -- write to control register
- CPU checks whether task is done -- read status register

Data Registers

- CPU transfers data to/from device



Device electronics

- performs actual operation
  - pixels to screen, bits to/from disk, characters from keyboard

I/O Device Examples

Device	Behavior	Partner	Data Rate (KB/sec)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Laser Printer	Output	Human	1,000
Graphics Display	Output	Human	30,000
Network-LAN	Input or Output	Machine	200-1,000,000
Internet	Input or Output	Machine	4,000- ?
CD-ROM (1x)	Storage	Machine	150
DVD-ROM (1x)	Storage	Machine	1,352
Magnetic Disk	Storage	Machine	100,000
Flash Memory	Storage-read	Machine	1,000-300,000
	Storage-write	Machine	1,000-10,000

Programming Interface

How are device registers identified?

- Memory-mapped vs. special instructions

How is timing of transfer managed?

- Asynchronous vs. synchronous

Who controls transfer?

- CPU (polling) vs. device (interrupts)

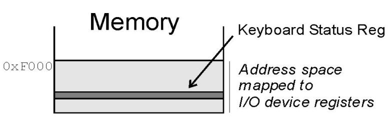
Memory-Mapped vs. I/O Instructions

Instructions

- designate opcode(s) for I/O
- register and operation encoded in instruction

Memory-mapped

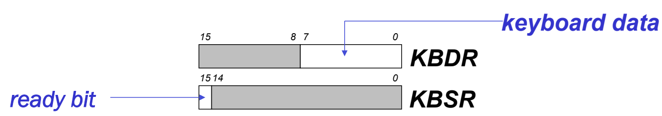
- assign a memory address
  to each device register
- use data movement
  instructions (LD/ST)
  for control and data transfer



Input from Keyboard

When a character is typed:

- its ASCII code is placed in bits [7:0] of KBDR
  (bits [15:8] are always zero)
- the "ready bit" (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored



When KBDR is read:

- KBSR[15] is set to zero
- keyboard is enabled

Transfer Control

Who determines when the next data transfer occurs?

Polling

- CPU keeps checking status register until
  new data arrives OR *device ready* for next data
- "Are we there yet? Are we there yet? Are we there yet?"

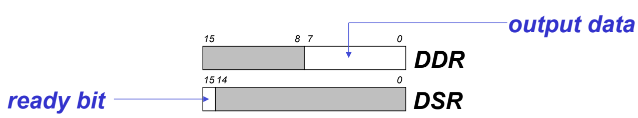
Interrupts

- Device sends a special signal to CPU when
  new data arrives OR *device ready* for next data
- CPU can be performing other tasks instead of polling device.
- "Wake me when we get there."

Output to Monitor

When Monitor is ready to display another character:

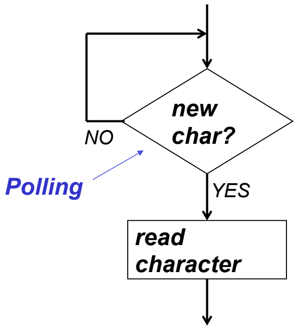
- the "ready bit" (DSR[15]) is set to one



When data is written to Display Data Register:

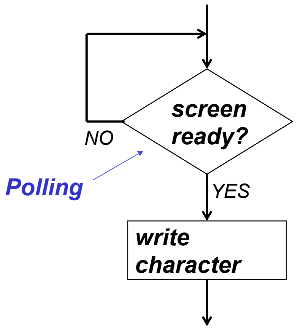
- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored
  (while DSR[15] is zero)

Basic Input Routine



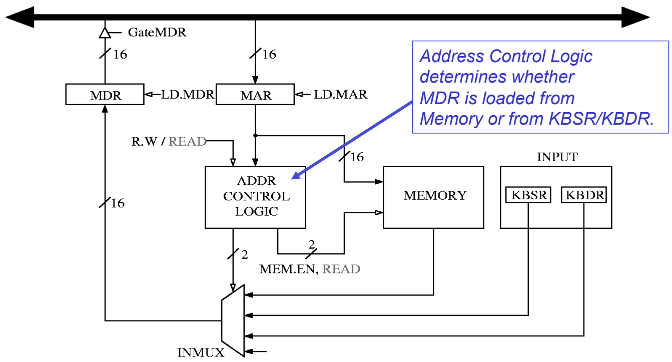

```
POLL     LDI  R0, KBSRPtr
         BRzp POLL
         LDI  R0, KBDRPtr

         ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```


Basic Output Routine

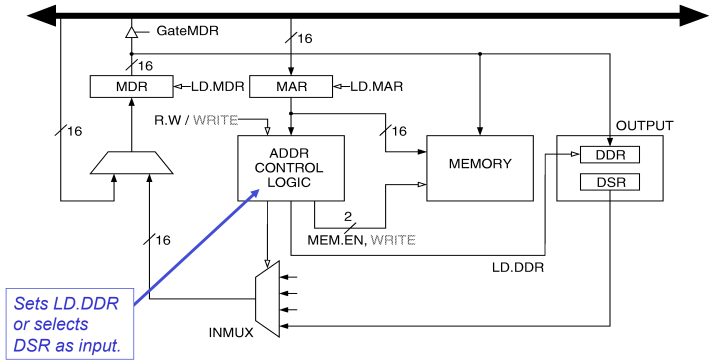



```
POLL    LDI  R1, DSRPtr
        BRzp POLL
        STI  R0, DDRPtr

        ...

DSRPtr .FILL xFE04
DDRPtr .FILL xFE06
```


Simple Implementation: Memory-Mapped Input



Simple Implementation: Memory-Mapped Output

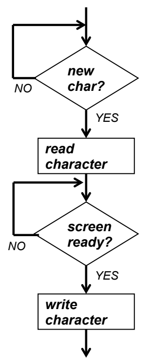


Keyboard Echo Routine

Usually, input character is also printed to screen.

- User gets feedback on character typed and knows its ok to type the next character.

```
POLL1   LDI  R0, KBSRPtr
        BRzp POLL1
        LDI  R0, KBDRPtr
POLL2   LDI  R1, DSRPtr
        BRzp POLL2
        STI  R0, DDRPtr

        ...

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
DSRPtr  .FILL xFE04
DDRPtr  .FILL xFE06
```



Interrupt-Driven I/O

External device can:

(1) Force currently executing program to stop;

(2) Have the processor satisfy the device's needs; and

(3) Resume the stopped program as if nothing happened.

Why?

- Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
- Example: Process previous input while collecting current input. (See Example 8.1 in text.)

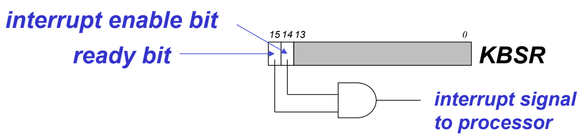
Interrupt-Driven I/O

To implement an interrupt mechanism, we need:

- A way for the I/O device to signal the CPU that an interesting event has occurred.
- A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.

Generating Signal

- Software sets "interrupt enable" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.



Priority

Every instruction executes at a selected level of urgency.

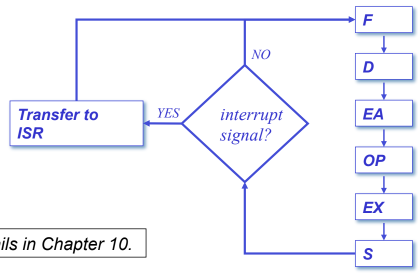
LC-3: 8 priority levels (PL0-PL7)

- Example:
  - Payroll program runs at PL0.
  - Nuclear power correction program runs at PL6.
- It's OK for PL6 device to interrupt PL0 program, but not the other way around.

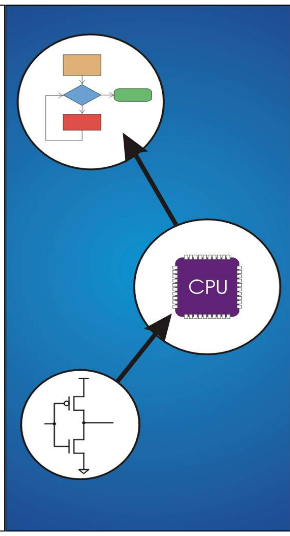
Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.

Testing for Interrupt Signal

CPU looks at signal between STORE and FETCH phases.
If not set, continues with next instruction.
If set, transfers control to interrupt service routine.



More details in Chapter 10.



Chapter 9
TRAP Routines and Subroutines

Computer Science 210 s1c
Computer Systems 1
2013 Semester 1
Lecture Notes

Lecture 19, 19Apr13:
Chap. 9: TRAP Routines & Subroutines

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

System Call

1. User program invokes system call.
2. Operating system code performs operation.
3. Returns control to user program.

In LC-3, this is done through the TRAP mechanism.

LC-3 TRAP Mechanism

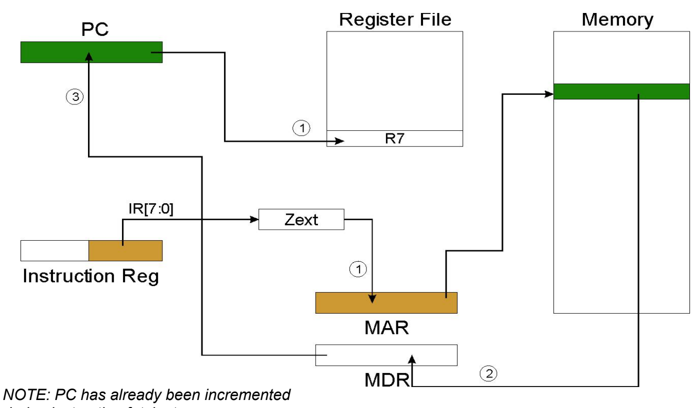
1. A set of service routines.
- part of operating system -- routines start at arbitrary addresses
  (convention is that system code is "below" x3000)
- up to 256 routines

2. Table of starting addresses.
- stored at x0000 through x00FF in memory
- called System Control Block in some architectures

3. TRAP instruction.
- used by program to transfer control to operating system
- 8-bit trap vector names one of the 256 service routines

4. A linkage back to the user program.
- want execution to resume
  immediately after the TRAP instruction

TRAP



NOTE: PC has already been incremented during instruction fetch stage.

TRAP Instruction

	15	14	13	12	11	10	9	8	7–0
TRAP	1	1	1	1	0	0	0	0	trapvect8

Trap vector
- identifies which system call to invoke
- 8-bit index into table of service routine addresses
  - in LC-3, this table is stored in memory at 0x0000 – 0x00FF
  - 8-bit trap vector is zero-extended into 16-bit memory address

Where to go
- lookup starting address from table; place in PC

How to get back
- save address of next instruction (current PC) in R7

RET (JMP R7)

How do we transfer control back to instruction following the TRAP?

We saved old PC in R7.
- JMP R7 gets us back to the user program at the right spot.
- LC-3 assembly language lets us use RET (return) in place of "JMP R7".

Must make sure that service routine does not change R7, or we won't know where to return.

Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 20, 30Apr13:

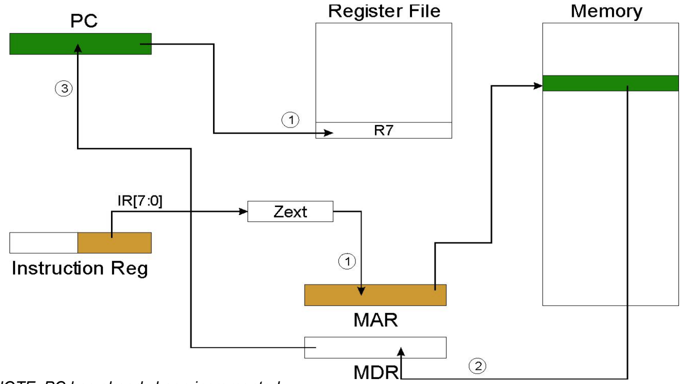
Subroutines; and Finally Ch. 10: The Stack

James Goodman

Department of Computer Science

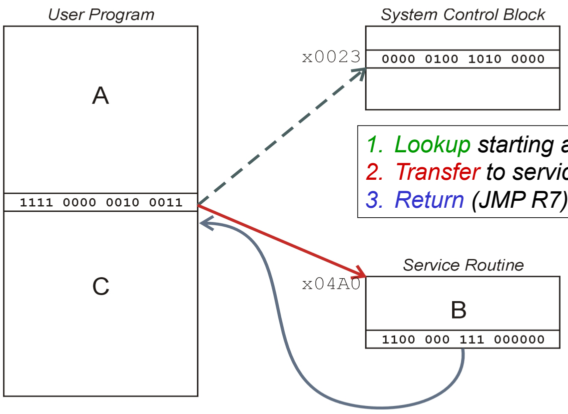
Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

TRAP



NOTE: PC has already been incremented during instruction fetch stage.

TRAP Mechanism Operation



1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).

Saving and Restoring Registers

Must save the value of a register if:
- Its value will be destroyed by service routine, and
- We will need to use the value after that action.

Who saves?
- caller of service routine?
  - knows what it needs later, but may not know what gets altered by called routine
- called service routine?
  - knows what it alters, but does not know what will be needed later by calling routine

Saving and Restoring Registers

Called routine -- *"callee-save"*
- Before start, save any registers that will be altered (unless altered value is desired by calling program!)
- Before return, restore those same registers

Calling routine -- *"caller-save"*
- Save registers destroyed by own instructions or by called routines (if known), if values needed later
  - save R7 before TRAP
  - save R0 before TRAP x23 (input character)
- Or avoid using those registers altogether

Values are saved by storing them in memory.

JSR Instruction

	15	14	13	12	11	10–0
JSR	0	1	0	0	1	PCoffset11

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.
- saving the return address is called "linking"
- target address is PC-relative (PC + Sext(IR[10:0]))
- bit 11 specifies addressing mode (one opcode, two instructions)
  - if =1, PC-relative: target address = PC + Sext(IR[10:0])
  - if =0, register: target address = contents of register IR[8:6]

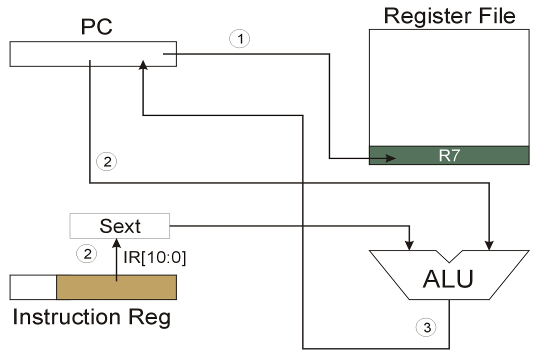
Subroutines

A subroutine is a program fragment that:
- lives in user space
- performs a well-defined task
- is invoked (called) by another user program
- returns control to the calling program when finished

Like a service routine, but not part of the OS
- not concerned with protecting hardware resources
- no special privilege required

Reasons for subroutines:
- reuse useful (and debugged!) code without having to keep typing it in
- divide task among multiple programmers
- use vendor-supplied *library* of useful routines

JSR



NOTE: PC has already been incremented during instruction fetch stage.

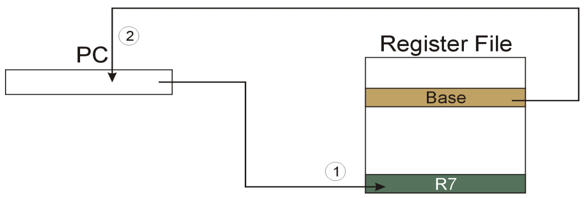
JSRR Instruction

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

Just like JSR, except Register addressing mode.
- target address is Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?

JSRR



NOTE: PC has already been incremented during instruction fetch stage.

Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.
- just like TRAP

Passing Information to/from Subroutines

Arguments
- A value passed in to a subroutine is called an argument.
- This is a value needed by the subroutine to do its job.
- Examples:
  - In 2sComp routine, R0 is the number to be negated
  - In OUT service routine, R0 is the character to be printed.
  - In PUTS routine, R0 is *address* of string to be printed.

Return Values
- A value passed out of a subroutine is called a return value.
- This is the value that you called the subroutine to compute.
- Examples:
  - In 2sComp routine, negated value is returned in R0.
  - In GETC service routine, character read from the keyboard is returned in R0.

Using Subroutines

In order to use a subroutine, a programmer must know:
- its address (or at least a label that will be bound to its address)
- its function (what does it do?)
  - NOTE: The programmer does not need to know *how* the subroutine works, but what changes are visible in the machine's state after the routine has run.
- its arguments (where to pass data in, if any)
- its return values (where to get computed data, if any)

Saving and Restoring Registers

Since subroutines are just like service routines, we also need to save and restore registers, if needed.

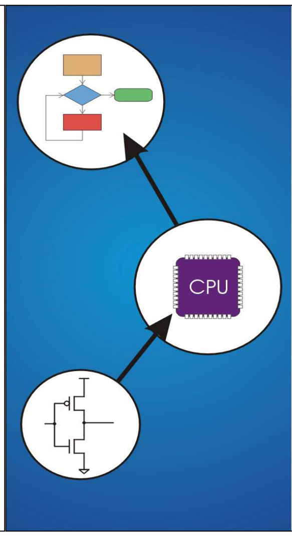
Generally use "callee-save" strategy, except for return values.
- Save anything that the subroutine will alter internally that shouldn't be visible when the subroutine returns.
- It's good practice to restore incoming arguments to their original values (unless overwritten by return value).

Remember: You MUST save R7 if you call any other subroutine or service routine (TRAP).
- Otherwise, you won't be able to return to caller.

Computer Science 210 s1c
Computer Systems 1
2013 Semester 1
Lecture Notes

The Stack

Credits: Slides prepared by Gregory T. Byrd, North Carolina State University



Chapter 10
And, Finally... The Stack

Stack: An Abstract Data Type

An important abstraction that you will encounter
in many applications.

We will describe Interrupt-Driven I/O
- The rest of the story…

Stacks

A LIFO (last-in first-out) storage structure.
- The first thing you put in is the last thing you take out.
- The last thing you put in is the first thing you take out.

This means of access is what defines a stack,
not the specific implementation.

Two main operations:
PUSH: add an item to the stack
POP: remove an item from the stack

Stack: An Abstract Data Type

An important abstraction that you will encounter
in many applications.

We will describe three uses:

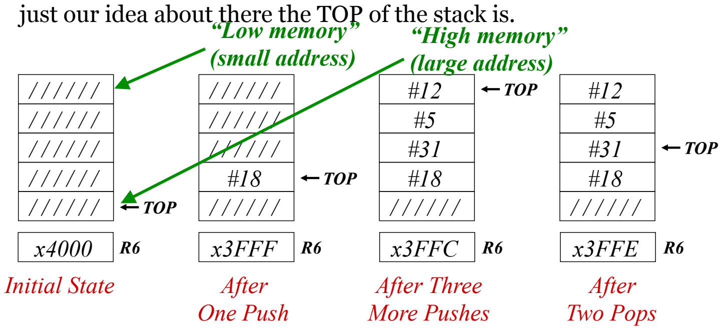
Interrupt-Driven I/O
- The rest of the story…

Evaluating arithmetic expressions
- Store intermediate results on stack instead of in registers

Data type conversion
- 2's comp binary to ASCII strings

A Software Implementation

Data items don't move in memory,
just our idea about there the TOP of the stack is.



By convention, R6 holds the Top of Stack (TOS) pointer.

Basic Push and Pop Code

For our implementation, stack grows downward
(when item added, TOS moves closer to 0)

Push

```
ADD  R6, R6, #-1 ; decrement stack ptr
STR  R0, R6, #0  ; store data (R0)
```

Pop

```
LDR  R0, R6, #0  ; load data from TOS
ADD  R6, R6, #1  ; decrement stack ptr
```


Computer Science 210 s1c

Computer Systems 1

2013 Semester 1

Lecture Notes

Lecture 21, 2May13:

The Stack: Interrupt-Driven I/O

James Goodman

Department of Computer Science

Credits: "McGraw-Hill" slides prepared by Gregory T. Byrd, North Carolina State University

Preview of C: Stack Frames

Major support issues posed by subroutines
- Linkage (how to get there and back)
- Passing parameters (where are they)
- Providing storage for local use (finding unique space for each invocation)

An *activation record* is a memory template of fixed size, allocated atomically as part of invoking a subroutine
- It allocates space to save parameters
- It provides storage for variables defined in the subroutine
- It provides a place for saving the return path.

A stack of *activation records* is an efficient way to address all three issues

By convention, register R6 is used as the stack frame pointer.

Interrupt-Driven I/O (Part 2)

Interrupts were introduced in Chapter 8.
1. External device signals need to be serviced.
2. Processor saves state and starts service routine.
3. When finished, processor restores state and resumes program.

Interrupt is an unscripted subroutine call, triggered by an external event.

Chapter 8 didn't explain how (2) and (3) occur,
because it involves a stack.

Now, we're ready…

Processor State

What state is needed to completely capture the state of a running process?

Processor Status Register

- Privilege [15], Priority Level [10:8], Condition Codes [2:0]

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

Program Counter

- Pointer to next instruction to be executed.

Registers

- All temporary state of the process that's not stored in memory.

Where to Save Processor State?

Can't use registers.

- Programmer doesn't know when interrupt might occur, so she can't prepare by saving critical registers.
- When resuming, need to restore state exactly as it was.

Memory allocated by service routine?

- Must save state *before* invoking routine, so we wouldn't know where.
- Also, interrupts may be nested – that is, an interrupt service routine might also get interrupted!

Use a stack!

- Location of stack "hard-wired".
- Push state to save, pop to restore.

Supervisor Stack

A special region of memory used as the stack for interrupt service routines.

- Initial Supervisor Stack Pointer (SSP) stored in Saved.SSP.
- Another register for storing User Stack Pointer (USP): Saved.USP.

Want to use R6 as stack pointer.

- So that our PUSH/POP routines still work.

When switching from User mode to Supervisor mode (as result of interrupt), save R6 to Saved.USP.

Invoking the Service Routine – The Details

1. If Priv = 1 (user), Saved.USP ← R6, then R6 ← Saved.SSP.
2. Push PSR and PC to Supervisor Stack.
3. Set PSR[15] = 0 (supervisor mode).
4. Set PSR[10:8] = priority of interrupt being serviced.
5. Set PSR[2:0] = 0. [?]
6. Set MAR = x01vv, where vv = 8-bit interrupt vector provided by interrupting device (e.g., keyboard = x80).
7. Load memory location (M[x01vv]) into MDR.
8. Set PC = MDR; now first instruction of ISR will be fetched.

Note: This all happens between the STORE RESULT of the last user instruction and the FETCH of the first ISR instruction.

Returning from Interrupt

Special instruction – RTI – that restores state.

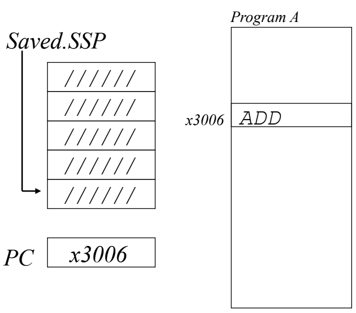
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1. Pop PC from supervisor stack. (PC = M[R6]; R6 = R6 + 1)
2. Pop PSR from supervisor stack. (PSR = M[R6]; R6 = R6 + 1)
3. If PSR[15] = 1, R6 = Saved.USP. (If going back to user mode, need to restore User Stack Pointer.)

RTI is a privileged instruction.

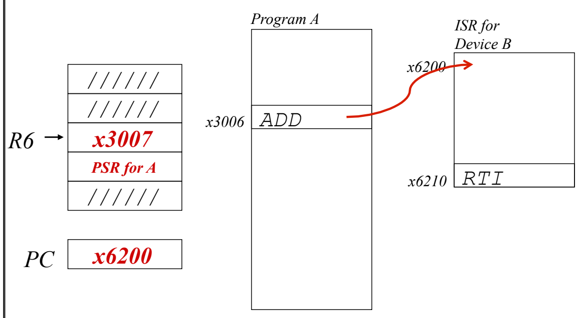
- Can only be executed in Supervisor Mode.
- If executed in User Mode, causes an *exception*. (More about that later.)

Example (1)



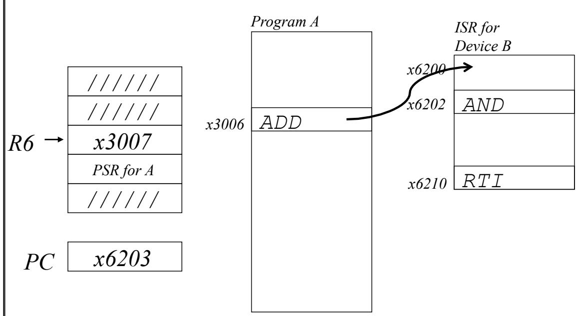
Executing ADD at location x3006 when Device B interrupts.

Example (2)



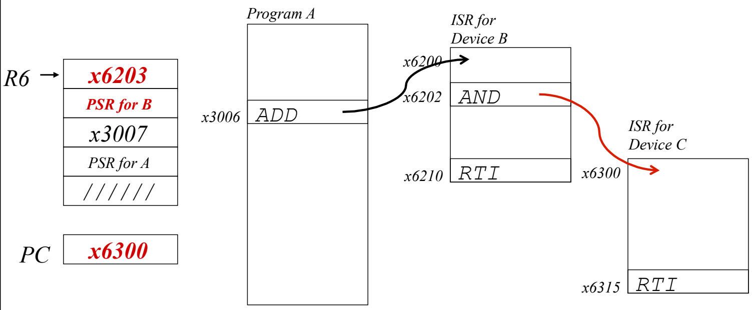
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to Device B service routine (at x6200).

Example (3)



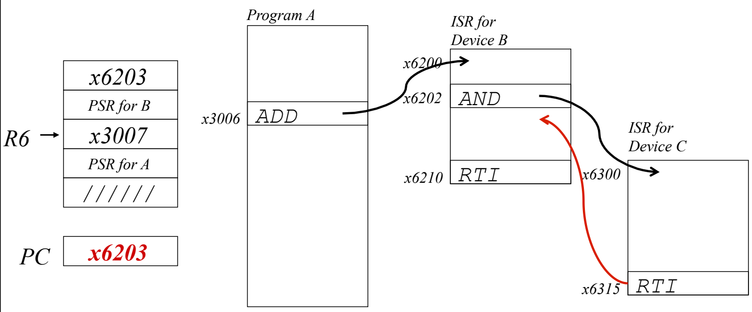
Executing AND at x6202 when Device C interrupts.

Example (4)



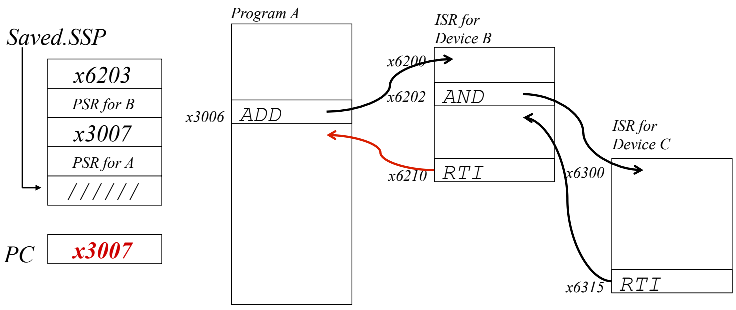
Push PSR and PC onto stack, then transfer to Device C service routine (at x6300).

Example (5)



Execute RTI at x6315; pop PC and PSR from stack.

Example (6)



Execute RTI at x6210; pop PSR and PC from stack. Restore R6. Continue Program A as if nothing happened.

Exception: Internal Interrupt

When something unexpected happens *inside* the processor, it may cause an exception.

Examples:

- Privileged operation (e.g., RTI in user mode)
- Executing an illegal opcode
- Divide by zero (or other forms of overflow)
- Accessing an illegal address (e.g., protected system memory)

Handled just like an interrupt

- Vector is determined internally by type of exception
- Priority is the same as running program