API - Application Programming Interface

Introduction to Computer Engineering

9 8 21

CS-EC 252

- Karu (CPU) → TA Daniel (Robotics) →
- TA _____ (Info Systems)

Smaller = More

→ programmable computers: Modify behavior
→ fixed function computers: Micro processors, (fixed chips)

Moore's Law: The time of transistor in. reduced every 6-24 months

Transistors: → source, drain, gate (can destroy bits)
↳ gate: allows current to flow from source to drain
↳ Bits / logic (0's & 1's) * End Gate *

Computers have gotten cheaper / faster - used to consume less power

Abstractions: "How do people program w/o knowing transistors?
↳ Interface & how

fixed function ← programmable →

Devices → circuits → microarchitecture → architecture (ISA) →
→ Compiler → logic
↳ ---> primitive language
↳ ---> finite state machine (adder, multiplier)

Assembly → representation of ISA.

* DEMIC → Atulya

Turing Machine: Universal Computer 9-13-2021

problem Statement
- Stated using "natural languages"
- may be ambiguous, imprecise

algorithm
- Step-by-step procedure, guaranteed to finish
- definiteness effective computability, finiteness (finite)

program
-

Instruction Set Architecture (ISA)

microarchitecture

logic circuits

devices

LC3 → Simple Computer

Bits, Data Types, and Operations

9 | 13 | 21

How to represent data on a computer

1. Presence of voltage "state 1"
2. Absence of voltage "state 0"

Computers use a binary digital system

digital: "0" ... "1"

analog: 0.5 ... 2.4 volts ... 2.9 volts

Bit: a basic unit of information

a collection of 2 bits has four possibilities $2^2 = 4$

a collection of 3 bits has eight possibilities $2^3 = 8$

2^n

Numbers: signed, unsigned, integer, floating point, complex... rational, rational,

Text: characters, strings

Images, sounds, lofted: pixels, colors, shapes

instructions: true, false

Data Type: representation / operations

→ Unsigned Integers:

weighted positional notation

"329"

represent 5 using string of ones ("11111")

2^2	2^1	2^0	
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

$$\begin{array}{r} 0\,1\,0\,1 \\ +\ 0\,0\,1\,0 \\ \hline 0\,1\,1\,1 \end{array}$$

10^2 | × 3 it's worth 300

10^1 | × 2 it's worth 20

10^0 | × 9 it's worth 9

Base 2 "101" = 5

$2^2 \times 4$

$2^1 \times 0$

$2^0 \times 1$

most sig. ... least sig.

msb: Most Sig. Bit

(continued) How many bits? | 9 | 13 | 21

Signed Integers:
with n bits. $= 2^n$ distinct values
- assign $\approx$ half to positive integers
1 thru 2^{n-1}

Two's complement: make arithmatic easy to do...

	00101 (5)
+	11011 (-5)
1	00000 (0)

ignore the carry out — "overflow"

∴ Write all the leading zeros
- The most sig. bit indicates pos./neg.
- If 1 then it is a neg. #

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

Range of an n-bit number:
-2^{n-1} thru $2^{n-1}-1$

Converting to Binary
1. If leading bit is 1 are take two's complement
2.
3.

- Conversion
- addition / subtraction
- floating Representation/floating
- ASCII

- You will know how many bits

9/15/21

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

DIVISION:
1. Divide by two
2. Keep quotient by 2
- write remainders right to left
3. append.

00100111 (two)
$2^5+2^2+2^1+2^0$

TOL

Representations
Operations: addition, subtraction
Logical Operators: and, OR, NOT, XOR

Addition

$$\begin{array}{r} 0\,1\,1\,0\,1\,0\,0\,0 \quad (104) \\ +\;1\,1\,1\,1\,0\,0\,0\,0 \quad (-16) \\ \hline 0\,1\,0\,1\,1\,0\,0\,0 \quad (88) \end{array} \qquad \begin{array}{r} 1\,1\,1\,1\,0\,1\,1\,0 \quad (-10) \\ +\;1\,1\,1\,1\,0\,1\,1\,1 \\ \hline 1\,1\,1\,0\,1\,1\,0\,1 \quad (-19) \end{array}$$

$$\begin{array}{r} 1\,2_{16} \\ 1\,5_{16} \\ \hline 1\,2_{16} \\ 1\,5_{16} \\ \hline 1\,0\,0 \end{array}$$

To get negative flip bits and add one

$$19 = 2^4 + 2^1 + 2^0 \qquad 16 + 2 + 1$$

$$1\,0\,0\,1\,1 \qquad 2^4\;2^3\;2^2\;2^1\;2^0$$

Subtraction = negate one number and add both together

Sign Extension: take sign bit and copy it into remaining unsigned bit

$$\begin{array}{r} 4\text{Bit} \\ 0\,1\,0\,0 \quad (4) \end{array} = \begin{array}{r} 8\text{bit} \\ 0\,0\,0\,0\,0\,1\,0\,0 \quad (4) \end{array}$$

Operators

A	B	A ^ B	A V B	A → ¬B	¬A	¬B
1	1	1	1		0	0
0	1	0	1		1	0
1	0	0	1		0	1
0	0	1	0		1	1

(Theorem proof)

Hexadecimal Notation (NOT a Representation)
↳ base-16 0-9 + A-F

Fractions (Fixed Point V. Floating)

$$0\,0\,1\,0\,1\,0\,0\,0.1\,0\,1 \quad (40.625)$$
$$1\,1\,1\,1\,1\,1\,1\,0.1\,1\,0 \quad (-1.25)$$

Scientific notation very large or small

16 → 8 → 23 → 6
S | exp. | fraction
$N = -1^S \times 1.\text{fraction}$

IEEE 754 (32-bit)

Imagine abstract reality - Supriano

Building Blocks of Computers

Transistor: Microprocessor contain MILLIONS of transistors

Intel pentium II 7 million

Compaq Alpha 21264 15 million

pentium III 28 million

NVIDIA AIGE 54 Billion

Logic: AND OR NOT

N-Type: When Gate has positive voltage (high) short circuit between #1 and #2 CLOSED

When Gate has zero voltage system (switch is open)

Gate = 1 on

Gate = 0 off

#1 & 2 terminal Source/Drain

NOT ^ OR

NOR gate

p-Type +2.9v When gate has positive gate is open when neg. gate is closed

Gate = 1 off

Gate = 0 on

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

inverter

OR gate

AND gate

NAND gate

NOR gate

NOT Gate: (inverter)

2.9 volts

In	Out
0	1
1	0

MOS: metal Oxide Semiconductor

CMOS: (complementary)

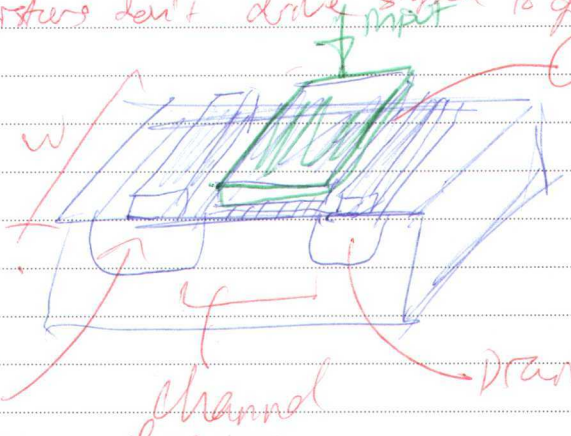
Gate, Source Dram

CS 555

9/24/21

- Terminal #1 ⏚
- Terminal #2
- Symbol GND: ⏚
- symbol VDD / POWER / HIGH VOLTAGE: VDD
- Resistor for N-TYPE
  Terminal #1 connected to output or Terminal #2

tip transistors don't drive signal to ground

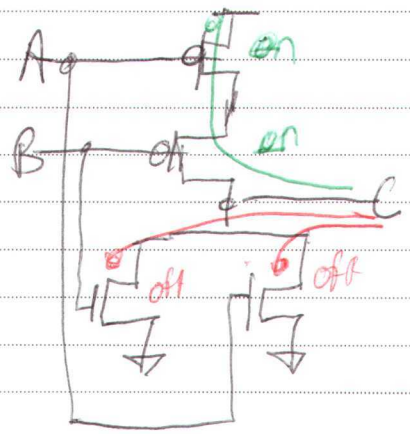


Typical Values	"0"	Illegal	"1"	
+5v, +3.3v, +2.9v, +1.1v, 0	0.5		2.4	2.9 volts

CMOS Circuits use both N-Type & P-Type

9/27/21

NOR Gate



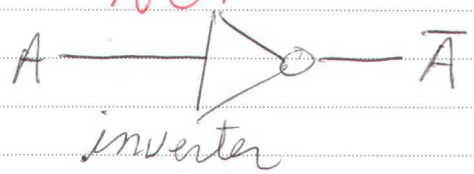
A = C
B = 0
C = 1

A A A

Practice Build a three input NOR Gate

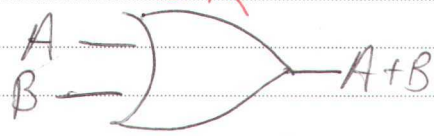
Basic logic gates

NOT

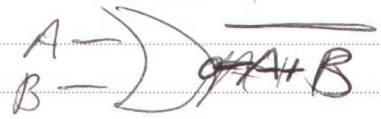


inverter

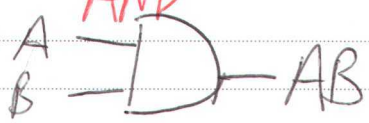
OR



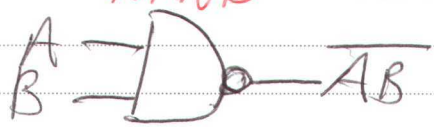
NOR

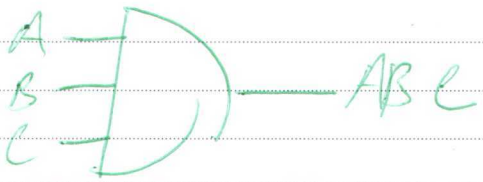


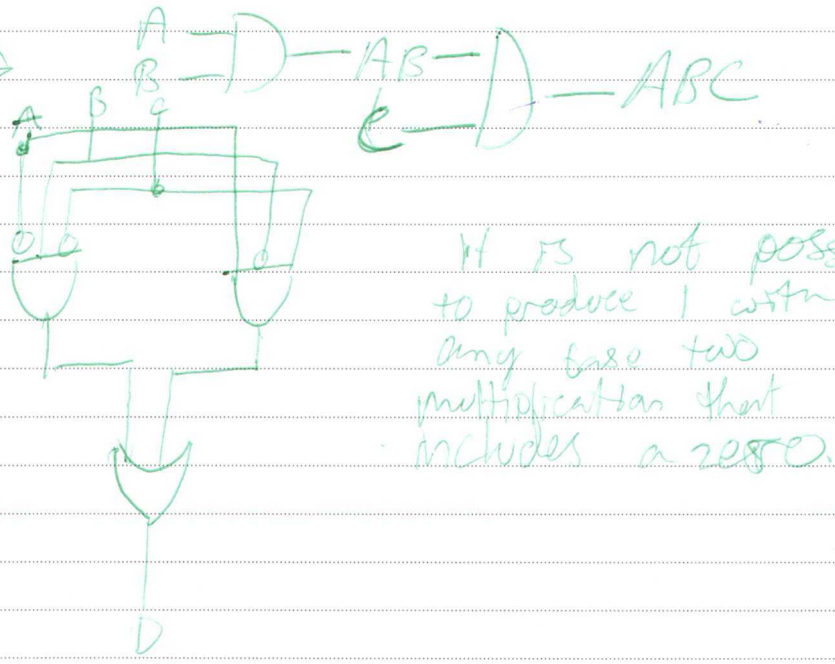
AND



NAND







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

It is not possible to produce 1 with any base two multiplication that includes a zero.

Remember De Morgan's Law

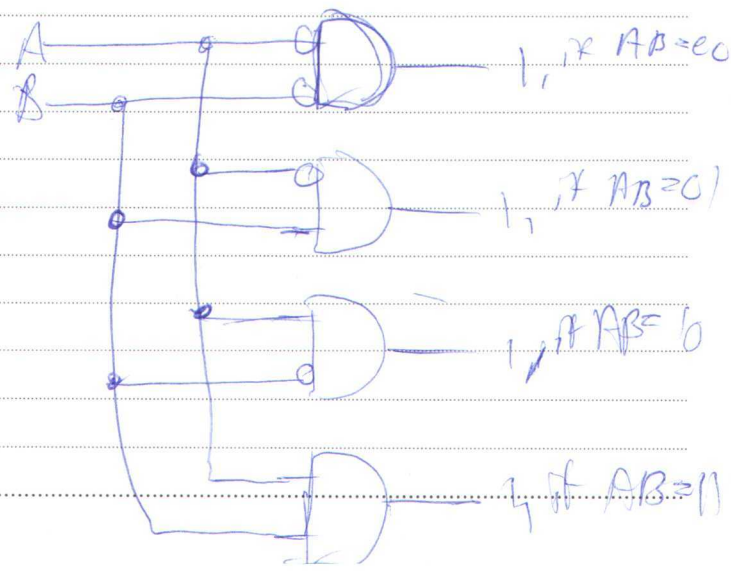
$A+B \rightarrow \overline{\bar{A}+\bar{B}}$

$\neg(A \wedge B) \rightarrow A \vee \neg B$

De Coder

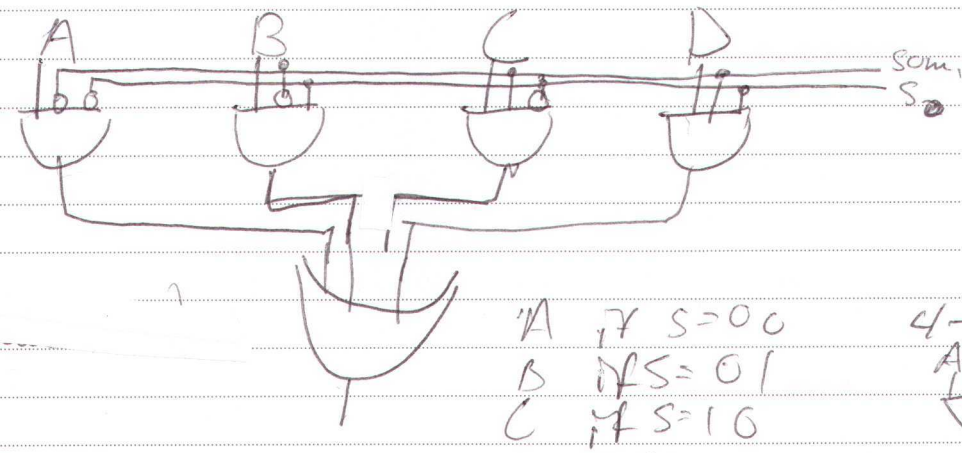
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

2bit



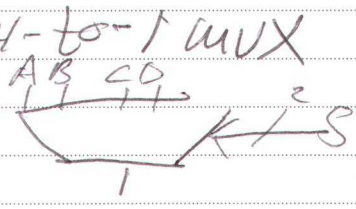
Multiplexer (Mux)

9 | 29 | 21



n-bit selector
2^n inputs one output

A if S=00
B if S=01
C if S=10
D if S=11



R-S latch // clocked R-S latch

R=S=1
- Hold current latch value

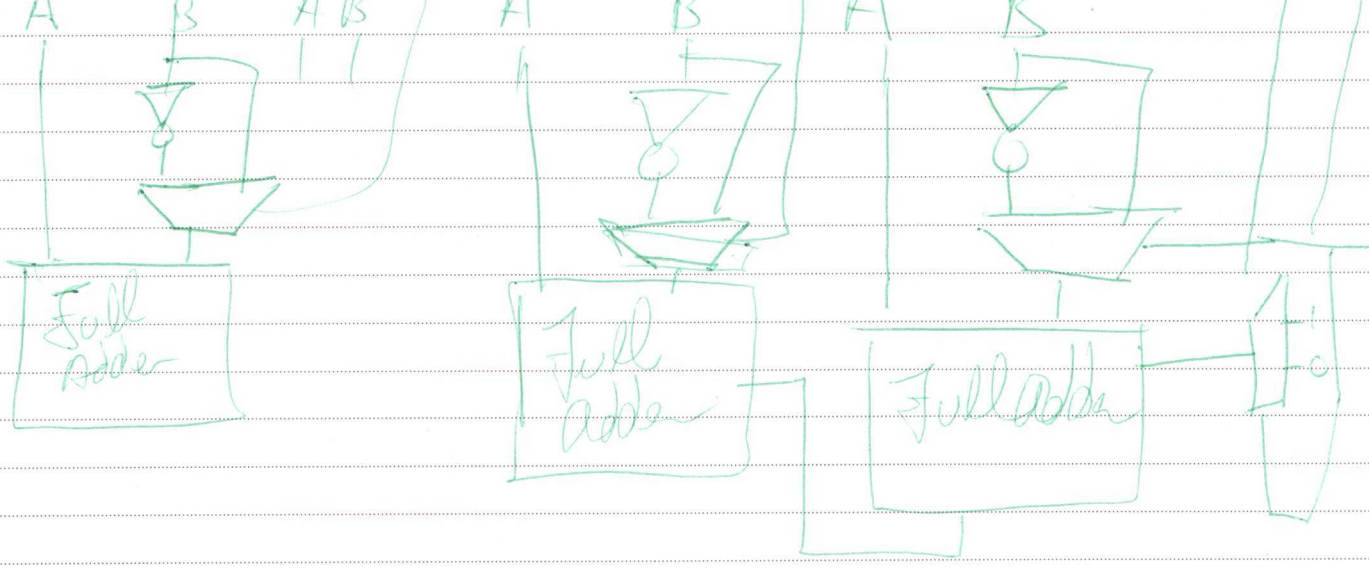
S=0 R=1
- set value to 1

R=0 S=1
- set value to 0

R=S=0
- both outputs equal one
- final state determined by electrical properties of gate
- Don't do this

Adder w/ Subtractor
↳ multiplexer

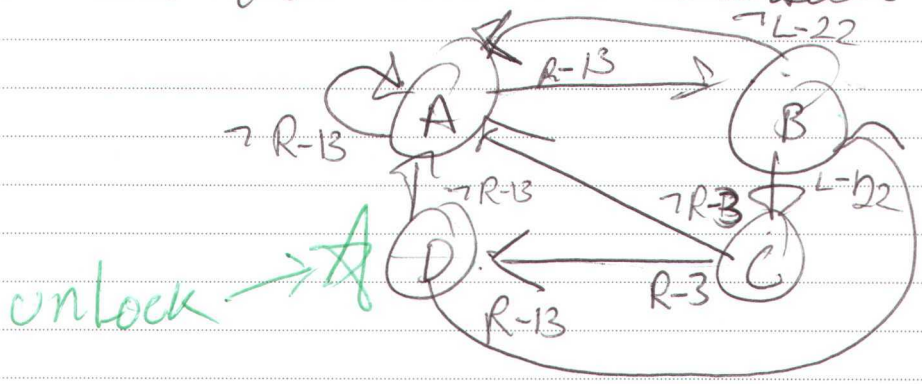
- Four-bit adder



State Machines

↳ The state of a system is a snapshot of all the relevant elements of the system at the moment the snapshot is taken.

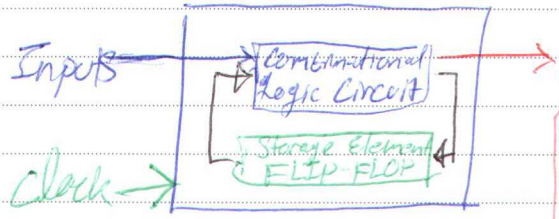
State Diagram: shows states and actions that cause a transition between states



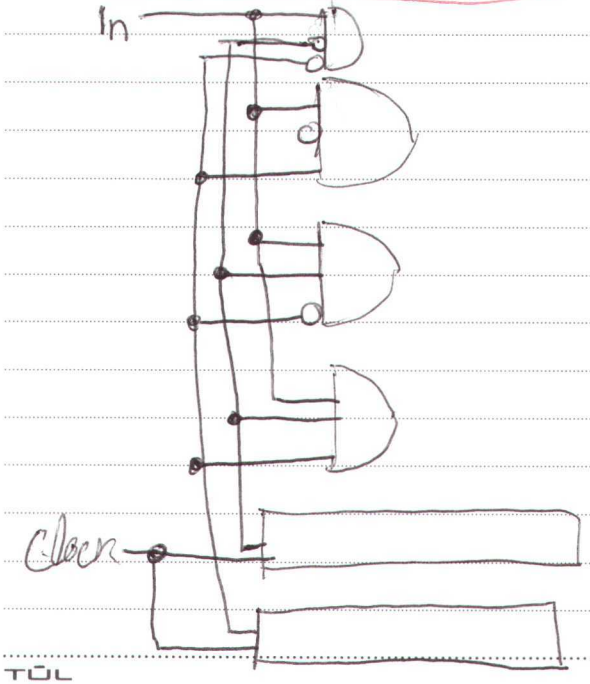
10/1/2021

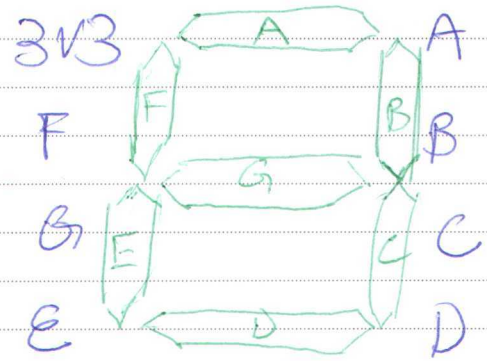
State Machine

* Capture new value at clock edge



Every flip-flop has an input/output





Review remember the range of 2's complement is always (-2^{n-1}) to $(2^{n-1}+1)$

example in 2 bits the range is

4-bit (-2^{4-1}) to $(2^{4-1}+1)$ or (-2^{2-1}) to $(2^{2-1}+1)$

or (-2^3) to (2^3+1) or (-2^1) to (2^1+1)

or (-8) to (7)

$1+2+4=7$

$-8+7=-1$

1000 1001

-2^3 $-2^3+2^2+2^1+2^0$

$-8+1=-1$

in 2's

$1111=-1$

Fixed Point Representation

111.111 6-bits total

$-1+.875 = \pm 1.25$

$1/2+1/4+1/8=$

example → 1.625

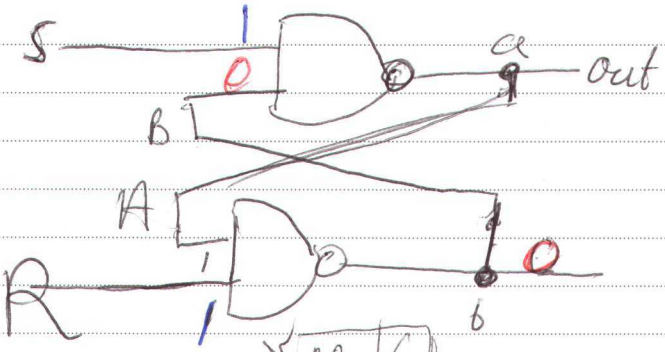
Floating Point

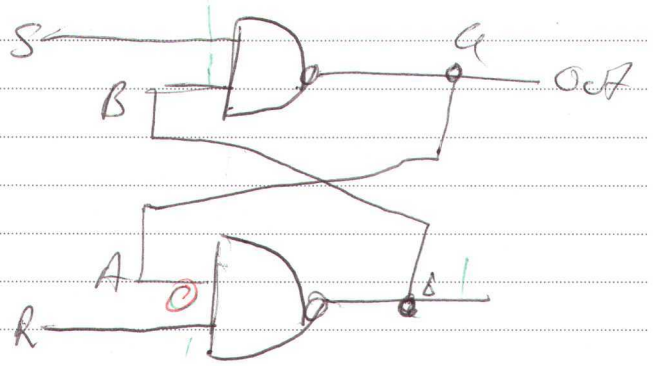
0 011 1 _

sign 8-bit Fraction 23-bits

R-S Latch Simple Storage Element

06.21





NAND Truth

AB	C
00	1
01	1
10	1
11	0

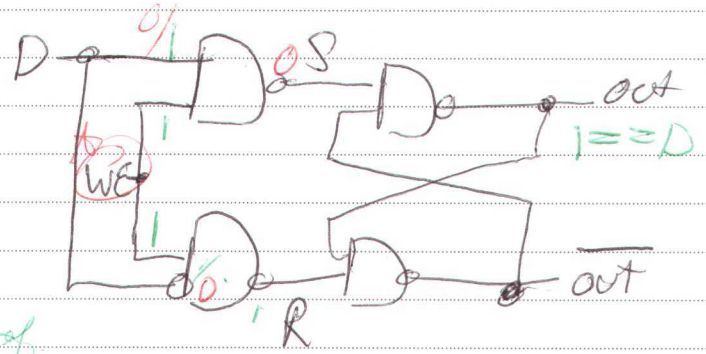
Summary

R=S=1
- Holds current state

S=0, R=1
- set to 1

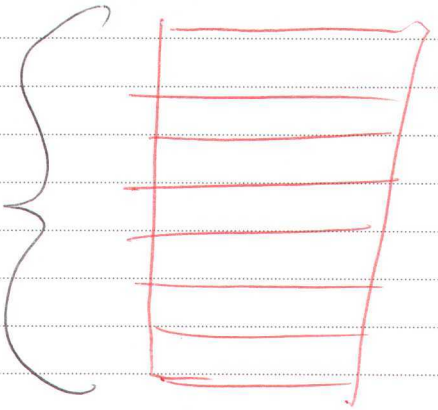
R=0, S=1
- set to 0

Gated D-Latch



A flip-flop is made of two D-latches

Memory (Single Register)



Address Space
# of locations $N = 2^n$
N usually powers of 2
Locations

addressability
# of bits per location

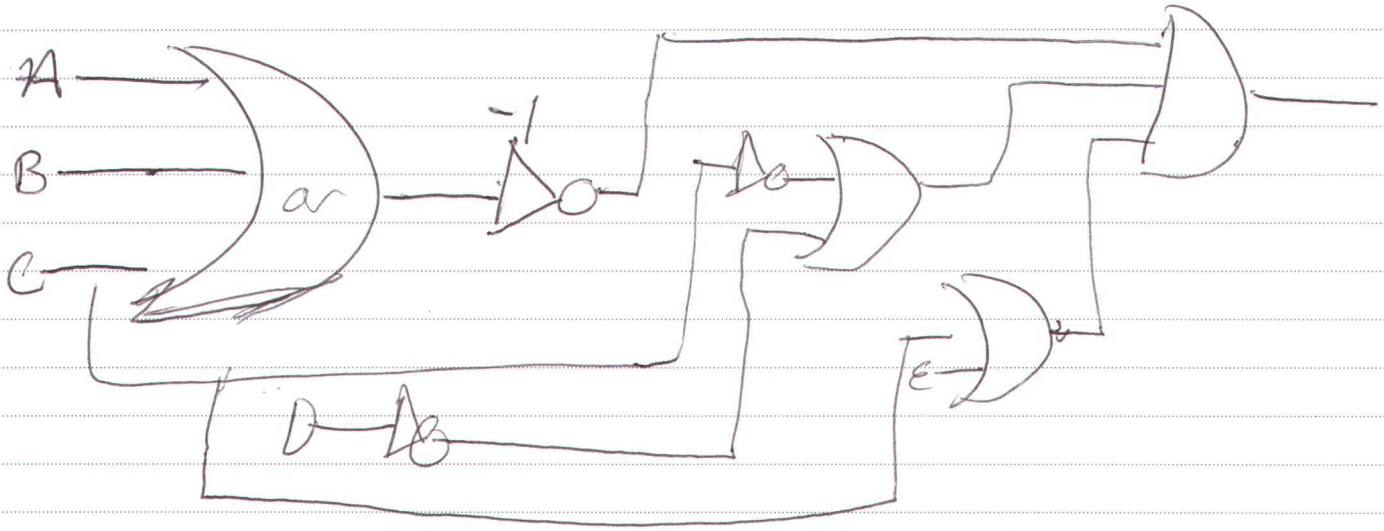
to find "least recently used"

* Write Enable = WE
Word select → Reading

Non-volatile storage (USB flash, Hard Drive)

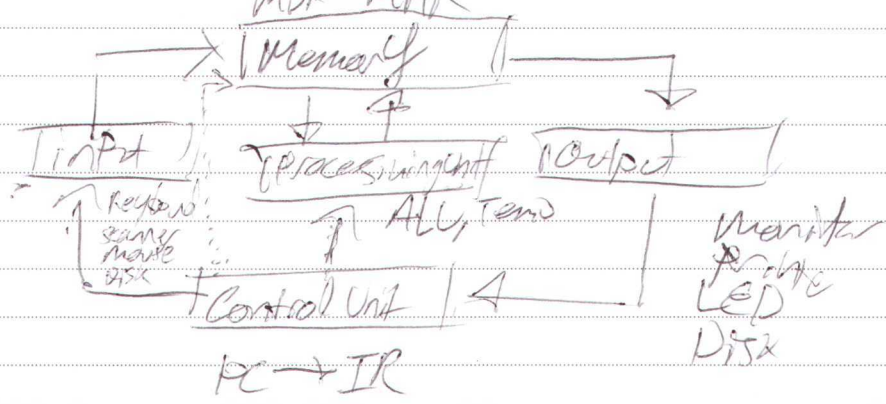
Boolean Logic to Logic Gates

$(\neg(A \vee B \vee C)) \wedge (\neg C \vee \neg D) \wedge (\neg E \vee C)$



The von Neuman Model (1943 – 1945)
- the 1st draft of a Report on EDVAC

memory, Processing Unit, Control Unit
created the concept of programmable computer



MAR
Memory Address Register

MDR
Memory Data Register

PC
Program Counter

IR
Instruction Register

ALU
Arithmatic Logical Unit

Memory = Address + Contents
- Unique n-bit identifier
- m-bit stored number

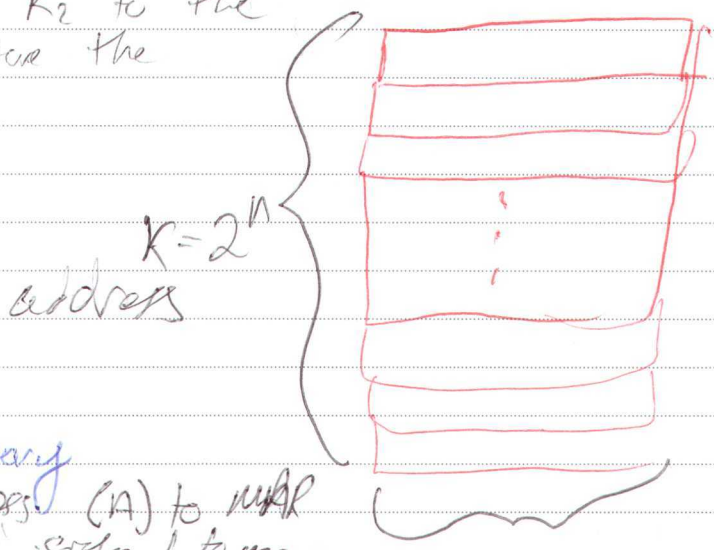
Basic Operations
- LOAD
- STORE
aka Read/Write

↳ not a stored program machine

↓
Atanasoff - Berry Computer

15 14 13 12	11 10 9	8 7 6	5	4 3	2 1 0
ADD	Dst	SRC-1	0	00	Src 2

"add the contents of R_2 to the contents of R6 and store the Result in R6."



Interface To Memory
- (1) write the address (A) to MAR
- (2) Send a "read" signal to memory
- (3) Read the data from MDR — n-bits

- (1) write the data (X) to the MDR
- (2) write the address (A) to the MAR
- (3) Send the write signal to memory

~~ALU~~ ALU - Arithmetic Logical Unit
functional unit - multiply, Square root, etc
Registers temp Storage 8: LC-2 · (R_0 ..... R_7)
word size LC-2 is 16-Bit

Input / Output } (Keyboard, mouse, scanner, DISK, Monitor, printer, LED, Desk)
LC-3 Supports input/output
- disk, network is both input / output

Control Unit
- IR — Instruction Register
- PC — program counter
- Control unit → reads an instruction from memory
  → interprets the instruction ...

Instruction — fundamental unit of work
- Opcode — operation to be performed
- Operand — detail / location to be used for operation

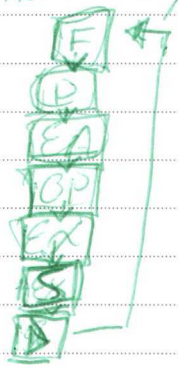
sequence
fetch → decode → Evaluate → fetch → execute → store!
in instruction from ... instruction ... address ... operands from ... operation performed ...

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Add				Dst			Src 1	0	0	0	Src 2					16	15	21

Von Neuman Model continued

6 clock cycles — Finish some instruction every clock cycle

Step 1: Fetch - instructions from memory
Step 2: Decode - Instructions
Step 3: Evaluate - address
Step 4: Fetch - operations from memory
Step 5: Execute - Operations
Step 6: Store - Results



Base + offset

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

0 1 1 0 | 0 1 0 | 0 1 1 | 0 0 0 1 1 0

add the value 6 to the contents of R3 to form a memory address. Load the contents stored in that address to R2.

ADD 0001 requires 3 operands (destination, two sources)
LDR 0110 requires 2 operands (goto / store to)
JMP 1100

MIPs vs MHz
- MHz → millions of clock cycle per second
- MIPs → million instructions Per Second

Assume a computer program has 5 instructions the clock frequency is 1 GHz. How

Assume a program with 20% type-1 instructions and 80% type-2 instructions:
Type 1 finishes in 5 cycles
Type 2 finishes in 7 cycles
assum the clock speed is 4 GHz

F-D-E-For-E-W each phase is one clock cycle.

= 1000 1110 1011 9089

10	19	21

ADD	Dst	Src1	0	00	Src2
0001	110	010	0	00	110

LDR	Dst	Base	offset
0110	010	011	000110

Register #	Value
R0	64
R1	31
R2	43
R3	1024
R4	2396
R5	1496
R6	9089
R7	1096

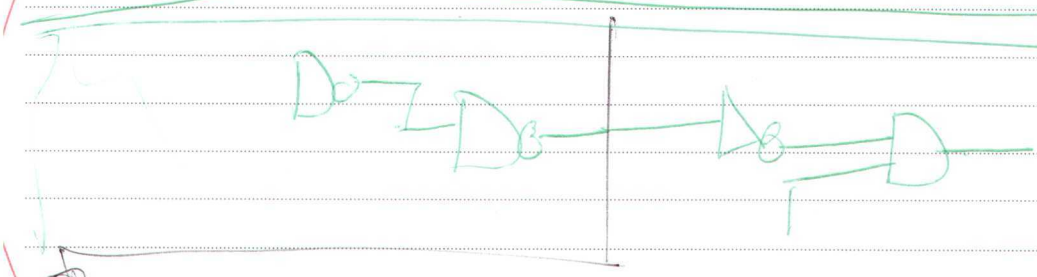
address	value
2048	64
1030	31
1024	43
...	1024
98	2396
96	1490
...	9089
0	1090

DST = R6
SRC1 = R2 (43)
SRC2 = R6 (9089) + = 9132

MAR = value(R3) + offset = 1030
-73

DST = R2
Base = R3
offset = 6

252 Special
MAR = value(DST) + offset
DST = MEMORY[MAR]



* Critical logic → less gates → faster transistors

Add → the access memories can be eliminated

Two Goals
- LC3 Instructions set which is the interface
  - Syntax of bits
  - Semantics - what a bit means
- LC3 datapath and how the machine works (microarchitecture)
- Some Programming

Instruction Set Architecture
ISA = All of the programmable-visible components and operations of the computer
- memory organization
  - address space - how many location 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
  - from a High Level language to machine language

LC3 Overview - Instruction Set
Memory - address space: 2^{16} location (16-bit addresses)
  - addressability: 16 bits
Registers
  - temporary storage, accessed in a single machine cycle (accessing memory generally takes longer than 1 cycle)
  - eight general purpose registers R0-R7
  - each 16 bits wide
  - how many bits to uniquely identify register
- Other registers
  - not directly addressable, but used by and affected by instructions
  - PC - (program counter)
  - condition codes

Opcodes
- 15 opcodes { 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)

TÜL

ALU - Arithmetic Logic Unit

(continued)

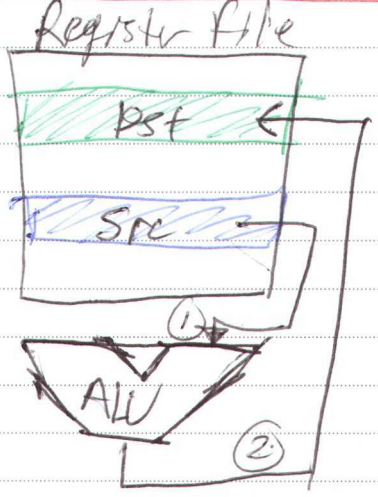
Data Types
- 16-bit 2's complement integer ** ↳ filling zero - zero extend (ZEXT) Rem: Sign extend (SEXT)

Addressing Modes
- How is the location of an operand specified?
  - non-memory address: immediate, register
  - memory address: PC-relative, indirect, base+offset
    - "... + to current address in program"

Operate Instructions
AND, ADD, NOT

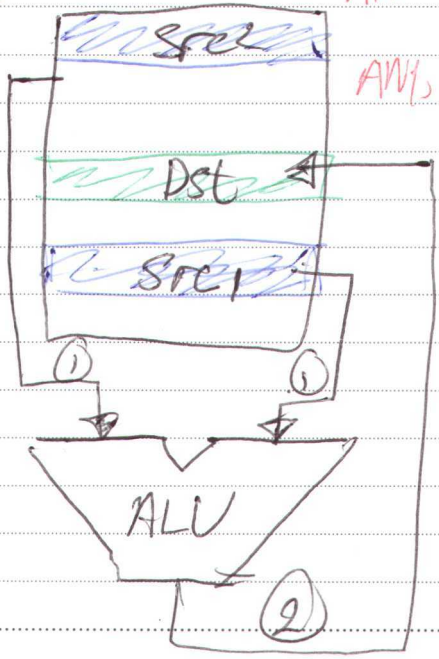
NOT

1001	DR(dst)	SR	111111



ADD / AND

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0	0	0	1	DST			SR1			0	0	0	SR2		
AND	0	1	0	1	DST			SR1			0	0	0	SR2		



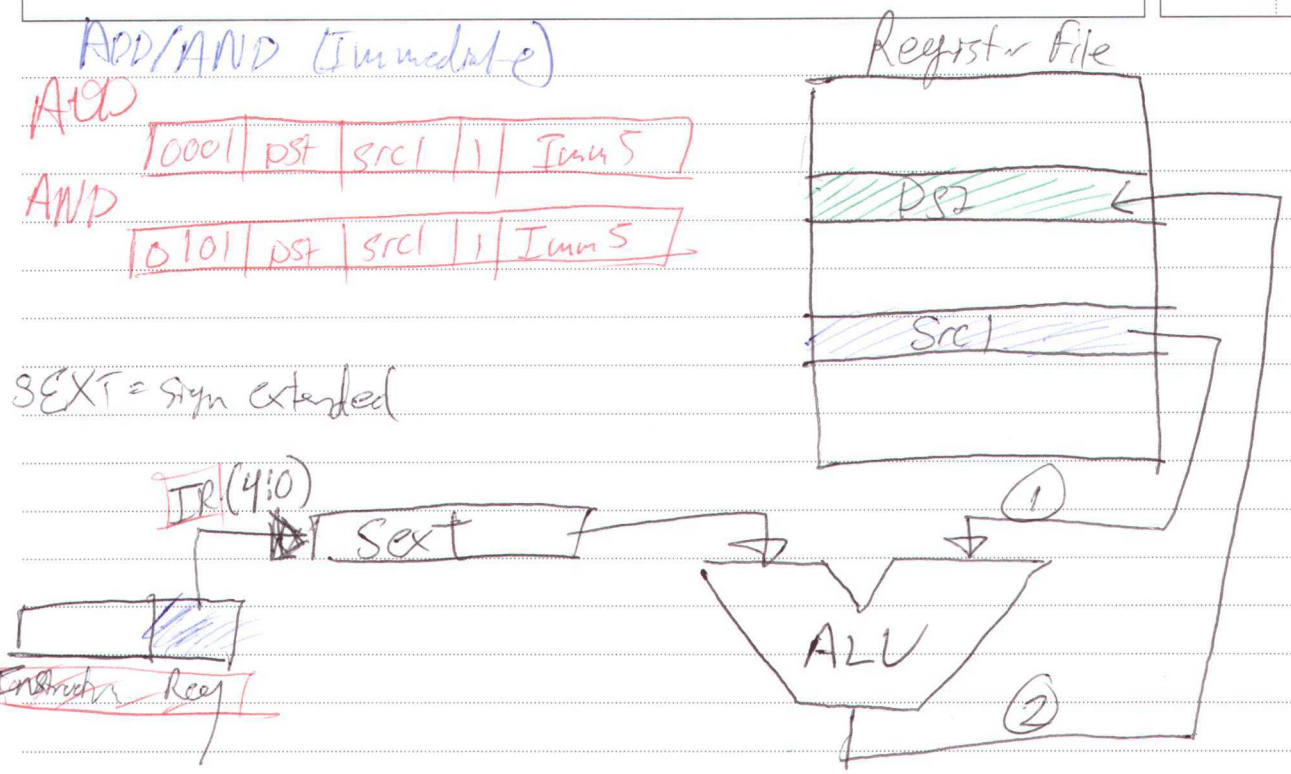
ADD/AND (Immediate)

ADD

| 0001 | DST | SRC1 | 1 | Imm5 |

AND

| 0101 | DST | SRC1 | 1 | Imm5 |



How to subtract? w/ only ADD, AND, NOT

→ (¬X ADD 1) ADD Y

How do we OR? "
DeMorgan's → ¬(¬A ∧ ¬B) = A ∨ B

How do we copy? from one reg to another

AND X with all 1's ADD, R_4, R_5, 0

How do we initialize to zero?
→ AND X with all zeros

Sign extension for immediate / offset is usually used

PC offset - 9-bit / 2's complement

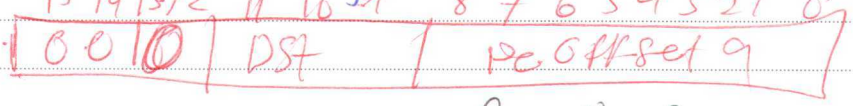
Data Movement Instructions

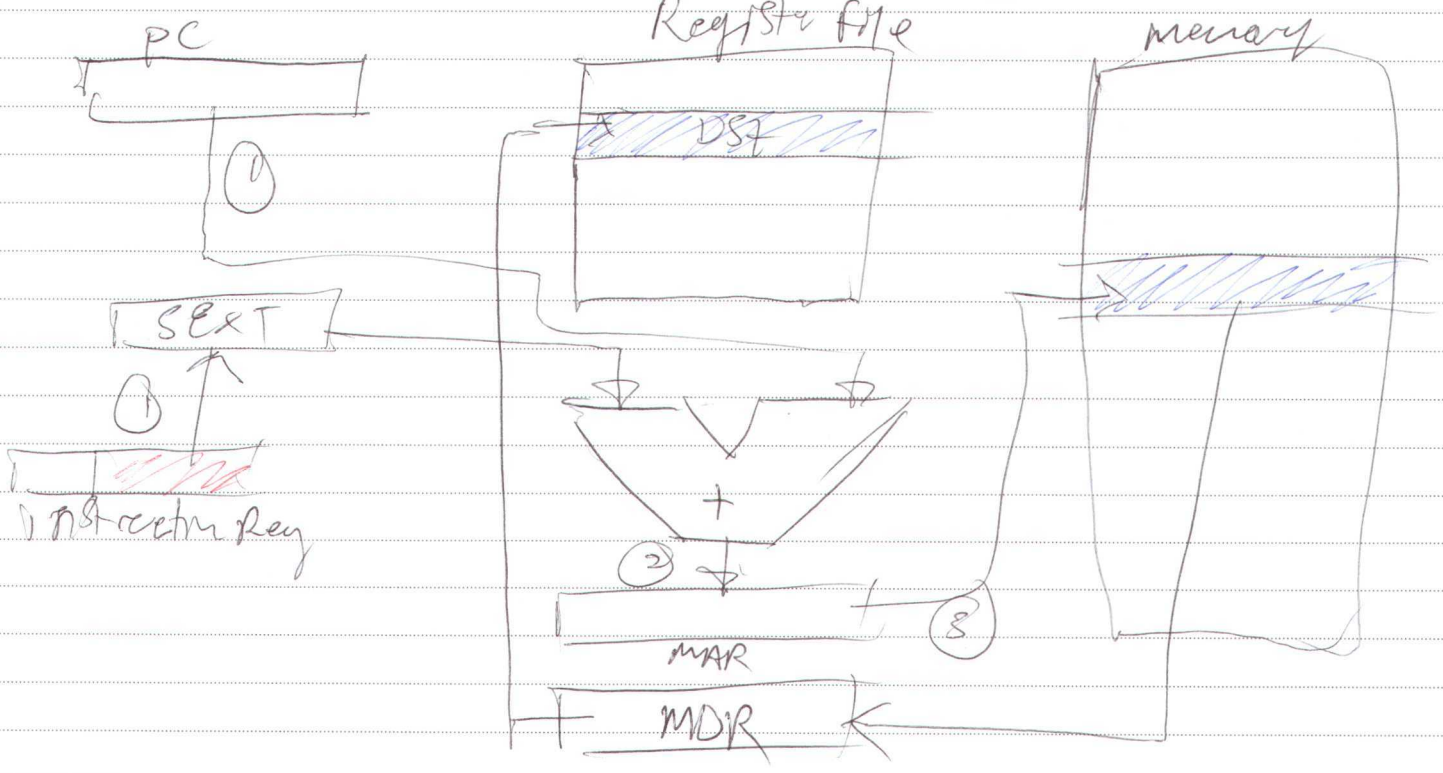
Load - read data from memory to register
- LD: PC-relative Mode
- LDR: base + offset mode
- LDI: Indirect mode

Store - write Data
- ST: PC-relative mode
- STR: base + offset
- STI: Indirect mode

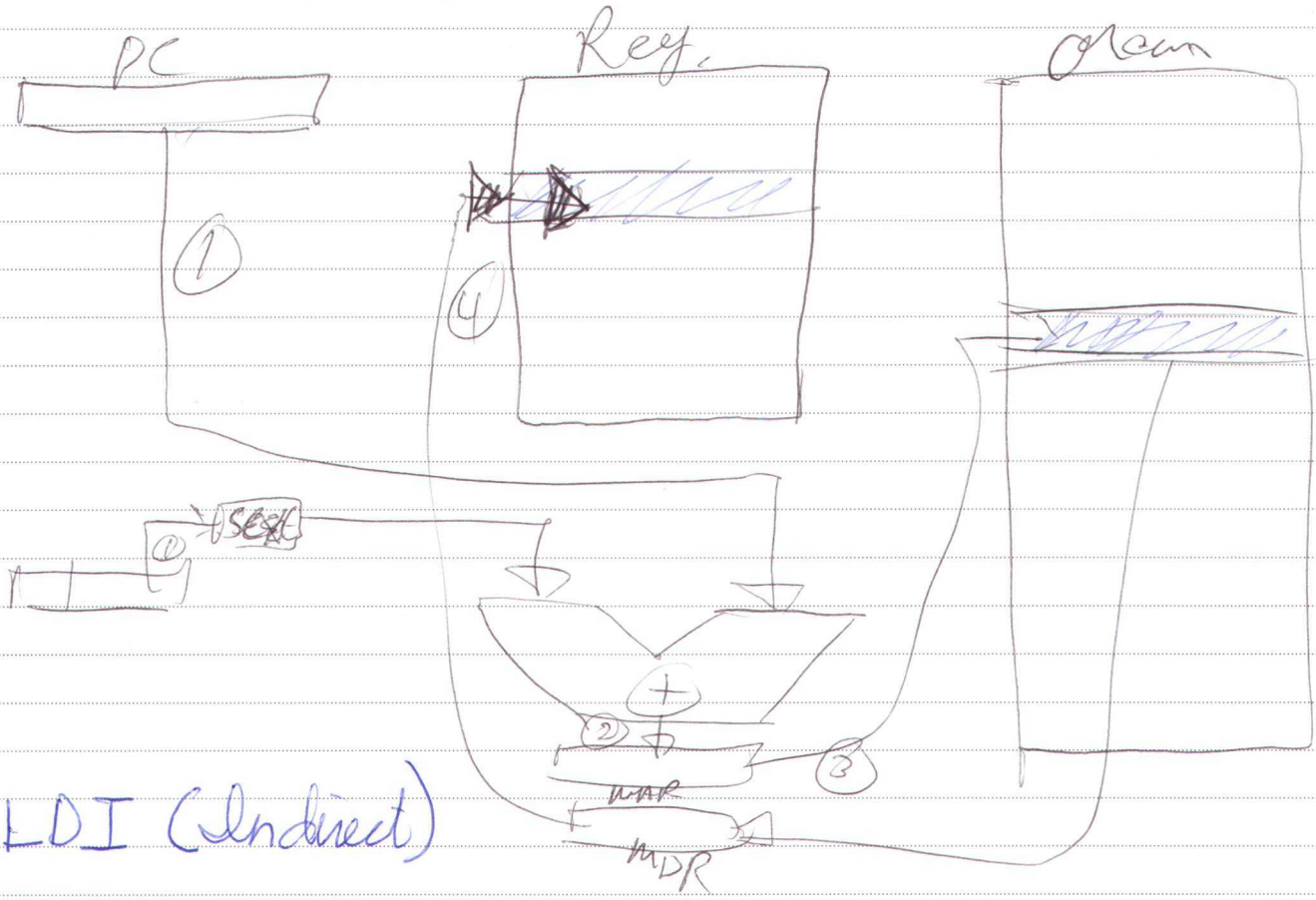
Load effective address — Compute address, save in register
- LEA: Immediate Mode
- Does not access memory

LD (PC-Relative)

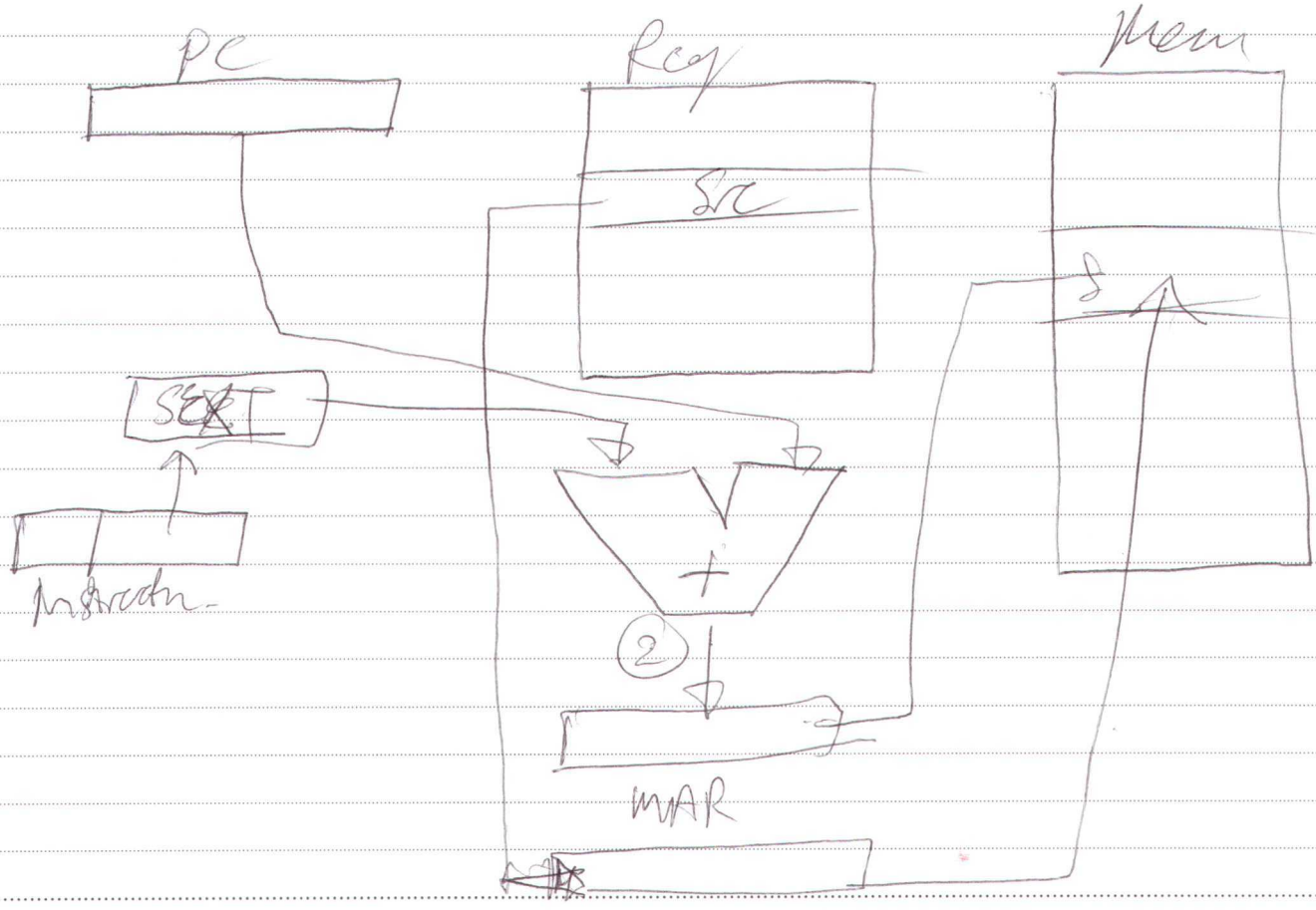




LD (PC-Relative)



LDI (Indirect)



Exam 2 (3/4) HW/Chapters

LSA Summary

10 25 21

NOT
ADD
AND
ADD
AND
LD
ST
LDI
STI
LDR
STR
LEA 1 1 1 0 DST PCOffset9 RF[DST] = PC+1+SEXT(PCOffset9)
BR N Z P

Control Instructions

Used to alter the sequence of instructions by changing the Program Counter

Conditional Branches

- 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 next sequential instruct.

Unconditional Branches

- always changes the PC

TRAP
- changes P.C to address of a OS
- routine will return control to next

Condition Codes

LC3 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 last instruction that altered a register

JMP
TRAP

TOL

Branch Instructions

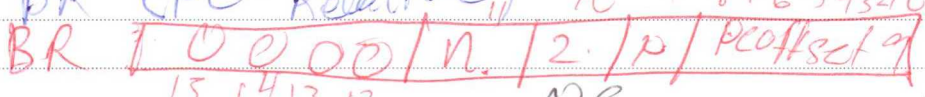
Branch specifies one or more condition codes.

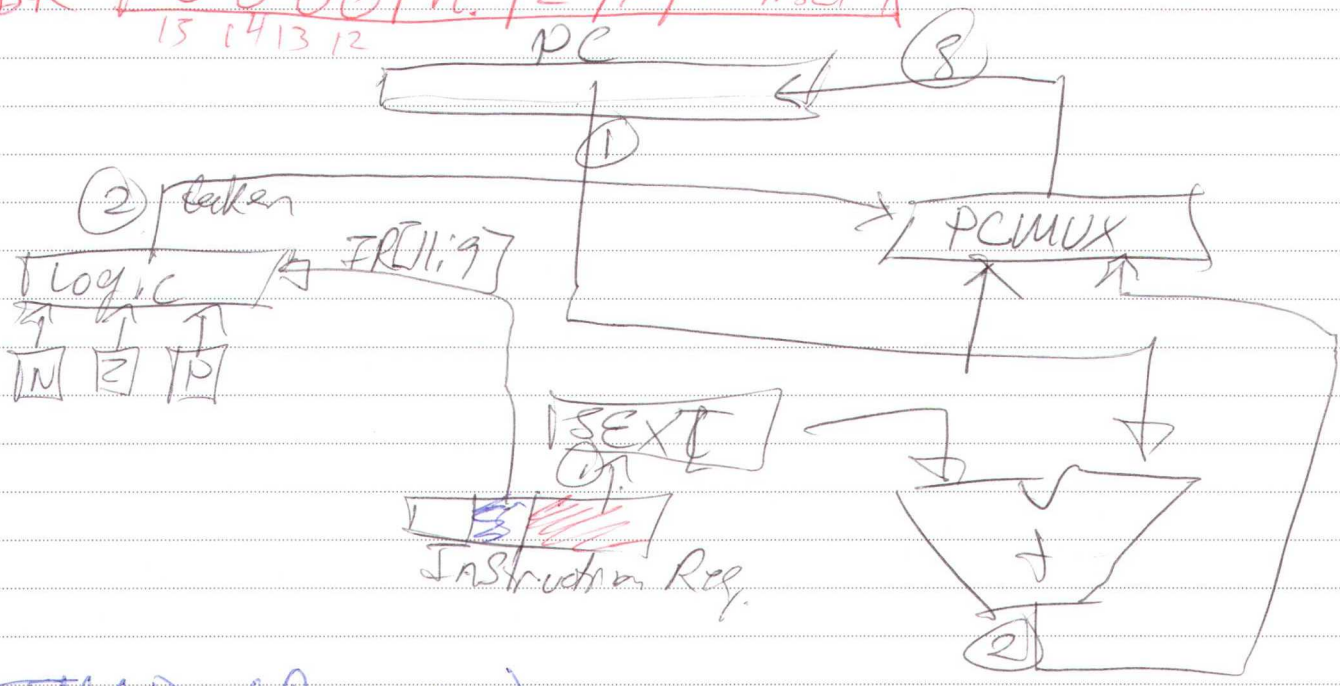
If the set bit is specified, the branch is taken

- PC-relative addressing:
  - is made by adding sign-extended offset (IR[8:0]) to current PC
  - Note: PC has already been incremented in FETCH-Stage
  - Note: Target must be within 256 words of BR-Instruction (9-Bits)

If the branch is not taken, the next sequential instruction is executed.

BR (PC-Relative)

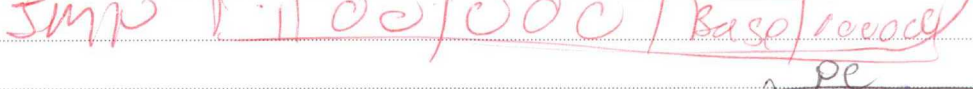


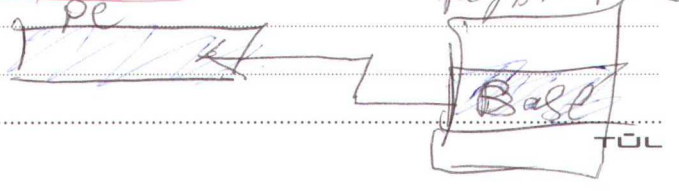


JMP (Register)

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

JMP

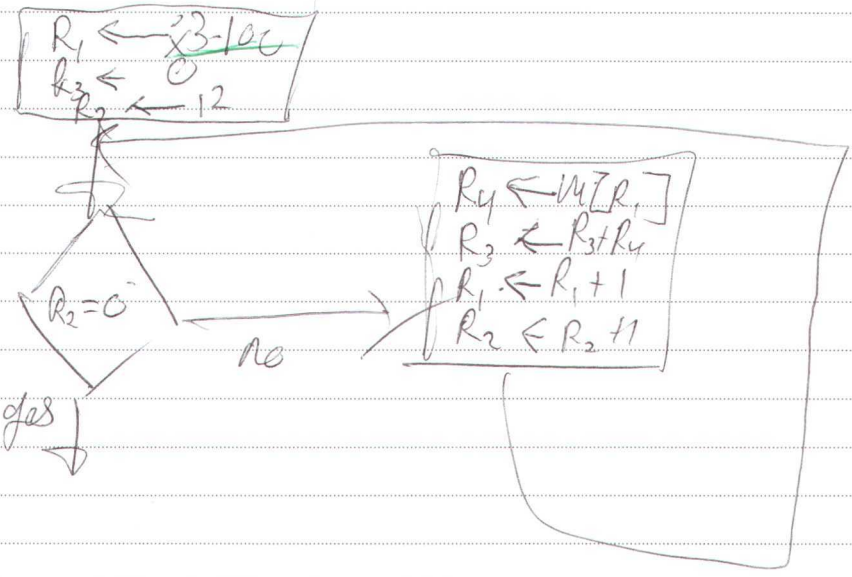




MDR, MAR, Memory, IR, PC, Reg File, SR1,in, SR2,out
clock elements.

Using Branch Instructions

Compute sum of 12 integers
numbers Start at locate x3100. Program starts at
location x3000.



Data path Components

- Global Bus
  - special set of wires that carry 16-bit signal
  - inputs to the bus are tri state devices
  - only one 16-bit signal enabled
  - any number of components can read the bus

Memory
- control and data regs for memory and I/O devices
- memory, MAR, MDR Also control signal for read/write

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.

Reg file
- two read address (SR1, SR2). one

16 ← 16 bit wire 2 ← 2 bit wire

TOL

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 (SR_1, SR_2), 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

PC and PCMUX
- Three inputs to PC controlled by PCMUX
  1. PC + 1 — FETCH stage
  2. Address adder — BR, JMP
  3. bus — TRAP

MAR and MARMUX
- Two inputs to MAR, controlled by MARMUX
  1. Address adder — LD/ST ; LDR/STR
  2. Zero-extended IR[7:0] — TRAP

Condition Code Logic
- Looks at value on the bus and generate 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 Machines
- 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 ect.

TOL

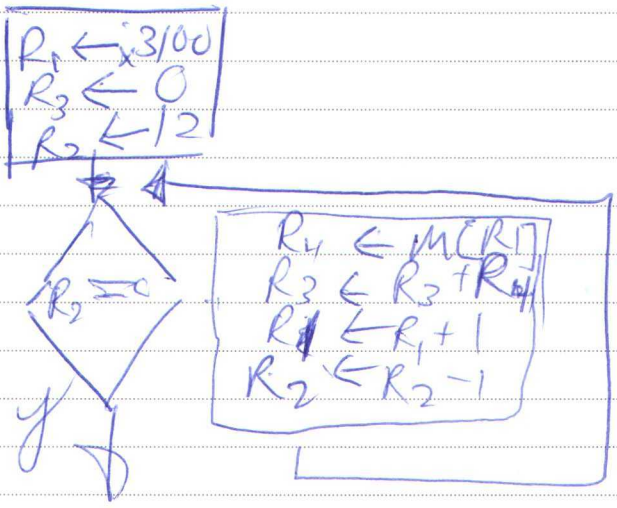
SR1, SR2, MDR ← Special from Control unit.

Mem = 1 R/W = 0 — ~~Memory~~ access ~~Memory~~ | 10 29 21

1-bit	2-bit	3-bit
LD.MDR		SR1
LD.MAR		SR2
LD.IR		DR
LD.PC		
LD.REG		

Read:
Store in ~~read~~ register value

5-34



Address		Instruction
x3000	1110	001
x3001	0101	011
x3002	0101	010
x3003	0001	010
x3004	0000	010
x3005	0110	100
x3006	0001	011
x3007	0001	001
x3008	0001	010
x3009	0000	111

2) Application, Program, Algorithms, Language - 11/1/2021

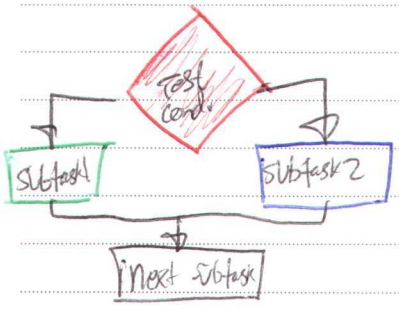
Problem Solving + Debugging

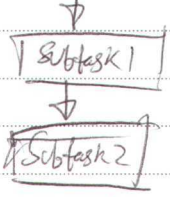
write instructions · Examine break points/looking at regs/mem.

"Systematic decomposition"

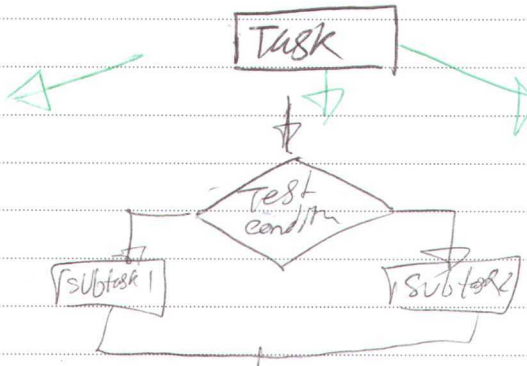
↳ decompose into smaller subtasks

Three Based Constructs

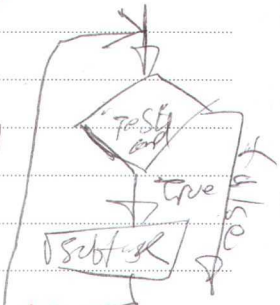




Sequential (procedural)



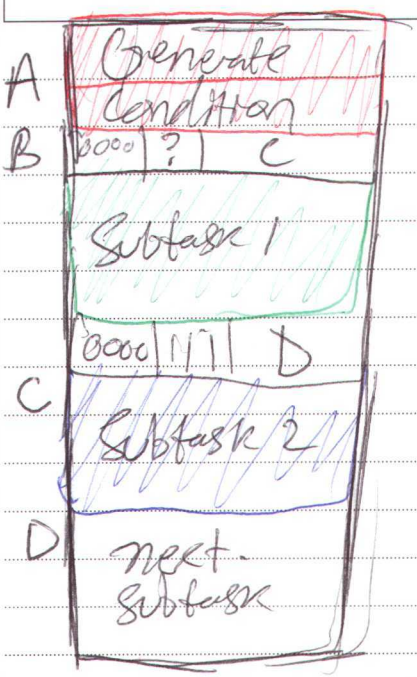
Conditional (if, then, else)



Iterative (for, while, do)

TÜL

Is the problem specified enough?



example

get char

count / print always at char

examine file

do A then do B sequential
if G then do H conditional
for each X, do Y iterative
do until W iterate

Same Successor Task

11/3/2021

Sequential instructions naturally flow . . .

Conditional / Iterative

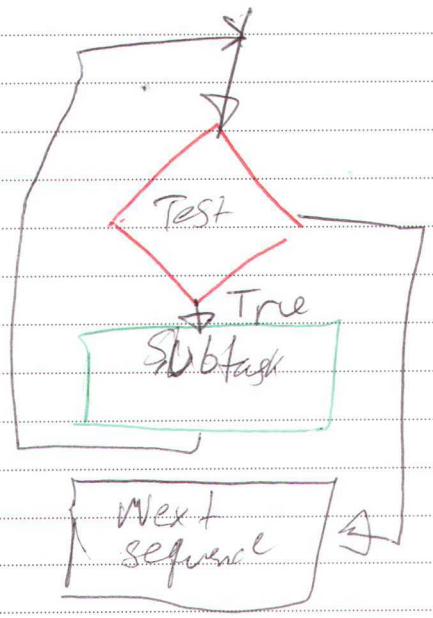
- Create code that converts into N, Z, or P

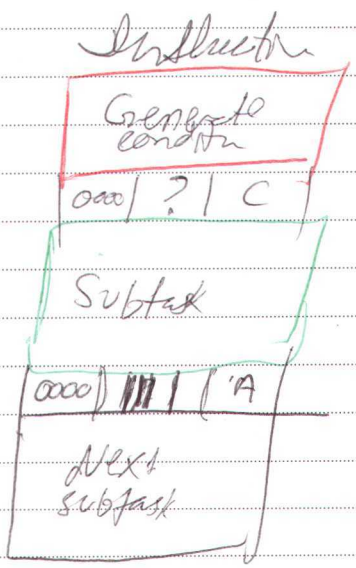
Example

e.g. $R_c = R1 * R2$

code: Subtract R1 from R2

if equal, Z bit will be set





Machine, assembler, compiler, WYSIWYG

Types of Errors

Syntax Error

* typing error
* almost any bit pattern corresponds to some legal instruction

Logic Errors

* program is ~~~~ legal but logically wrong

Data Errors

* input data is different than what was expected
* test the program with a variety of inputs

Tracing a Program

Single Stepping

* execute one instruction at a time
* → can be tedious but very useful

Break Point

* break at a specific point

Watch Point

* when a certain memory changes causes a break

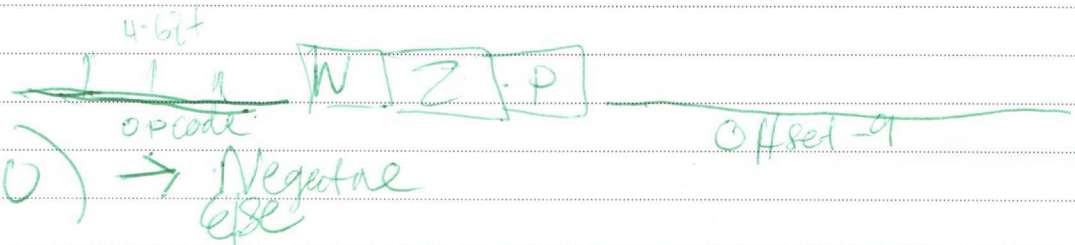
Cross Compiler

→ writing instruction set on other hardware

(1) - check logical negation

(1a) if false then ~~GOTO~~ Branch (pos is true)
BRnz

(1b) if true run Else Jump (pos is false)



0101 011 011 1 00000; ^R3, 0
0001 011 011 1 01010; ^R3, 10

if (R3 > 0)
R4 = 10

{ Destination
Load
or Store Memory

Chapter 7 | 9.2 11/9/2021

Assembly language & Implementation:
Computers see 0001 110 001 0000 10
Humans write ADD R6 R1 R2 ; R6 = increment index reg.
Assemble - a program that converts human instructions into computer's language (binary)
↑ file always starts w/ ".x"

.ORIG x3050 ← the dot is a directive to machine
↳ where the program originates

Label AGAIN
↳ Cannot begin w/ a dot "."
↳ Cannot be reserved string
↳ Cannot begin w/ a number

TOL

BR - "Branch" Specify N Z P
BRN Z, BRP, BRNZP ..... [-1] [0] [1]

offsets - (Rem(pc+1)) - # of instructions
+ 1 * -1 to go backward

```
AGAIN   ADD  R3, R3, R2        R3*R2=?
        ADD  R1, R1, #-1       for (int i=6, i>0, i--)
        BRP  AGAIN             { R3 + R2 ← R3
                                }
```

- BLKW 1 → clear.
- FILL x0006 → fill this location with.
- END

Each line is one of the following:
- instruction, assemble directive, a comment

Opcodes
ADD AND LD

Operands
- register
- number # (decimal) or x(hex) or b(binary)
- label

Label - placed at beginning, assigns symbolic name

Comment - anything after ";"
ignored by assembler

opcode	operand	
.ORIG	address	starting addr. of program
.END		ending of program
.BLKW	n	allocate n words to storage
.FILL	n	allocate one word, init. w/n value
.STRINGZ	n-character string	allocate n+1 locations, initialize w/characters and null term

TUL

LEA & LDR = 2 ways to access memory

TRAP codes - pseudo instructions

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
OUT	Trap x21	Write one character to console
GETC	Trap x20	Read one keystroke ← R0 ← [7:0]
PUTS	Trap x22	Write null-terminated string to console

Style — Guides

1 - have headers
2 - start labels
3 - use comments

11/10/2021

.ORIG	starting address
.END	
.BLKW	
.FILL	
.STRINGZ	n-character string ← Create escape if one char = x30

Example:

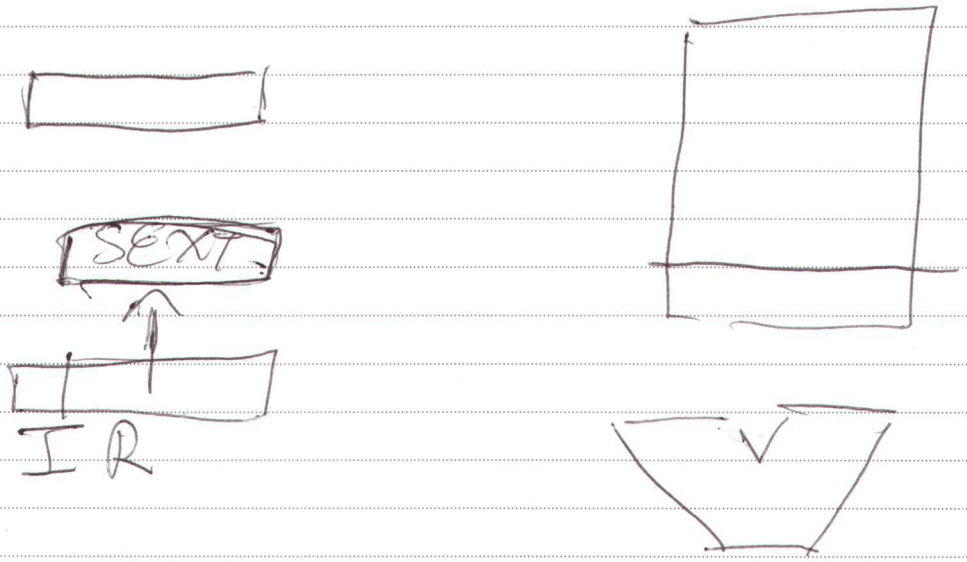
```
        .ORIG x3000
x3000   LEA R2, charprint1
x3001   LDR R0, R2, #0
x3002   OUT
x3003   LD R0, charprint2
x3004   OUT
x3005   charprint1 .FILL x0041
x3006   charprint2 .FILL x0042
```

CC	R0	R1	R2	R3	R4	R5	R6	R7	PC	IR	PSR

JSR Instruction 7-20

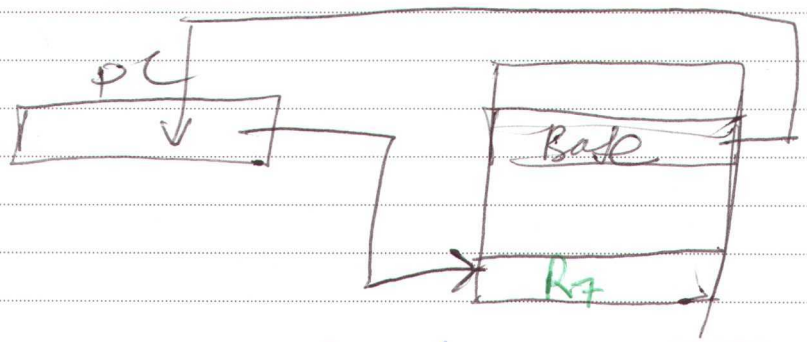
JSR 0100, PC offset11, Should write R7

Jumps to a location → like branch but does not require a condition



JSRR 0100 000 Base 000000 Should write R7

↑ Just like JSR but w/ address in (Restricted range)



Returning from a subroutine

RET (JMP R7) gets us back to calling routine ← just like JMP

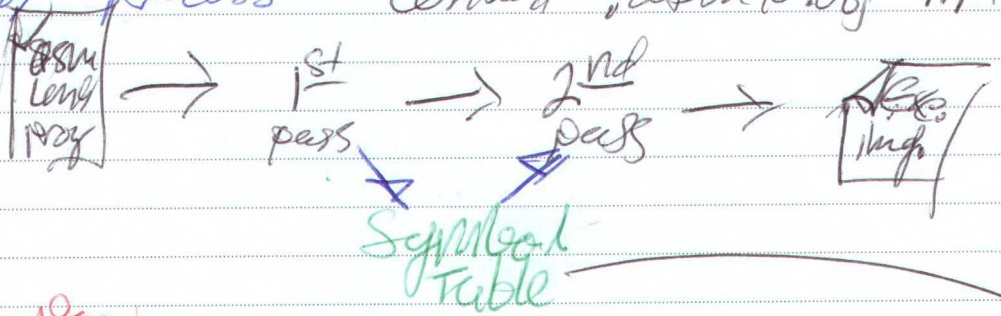
1100 000 111 000000

Write subroutine first

Using subroutines
- Address (label bound to its address)
- Function (What does it do?)
- Argument (where to pass data in)
- Return (where to get computed data)

"callee-save"
- save anything that the subroutine will alter internally
- It's good practice to restore incoming to their original value

Assembly Process — Convert .asm to .obj 11/5/2021



First Pass
- Scan program file
- find all label and calculate the address

Second pass:
- Convert instructions to machine language using information from Symbol Table

Loading + More sophisticated loader able to relocate
* must readjust branch target

Linking - Suppose define a symbol in one module.
* some notation such as .EXTERNAL used to tell a symbol is defined in another module
* linker will scan symbol table of another module.

saved register

Chapter 17 - wrap up code snippets

11 | 17 | 21

- [ ] read character
- [x] write character
- [ ] if-then-else
- [ ] while loop
- [ ] for loop
- [ ] sub-routine-count-down
- [ ] stash-reg

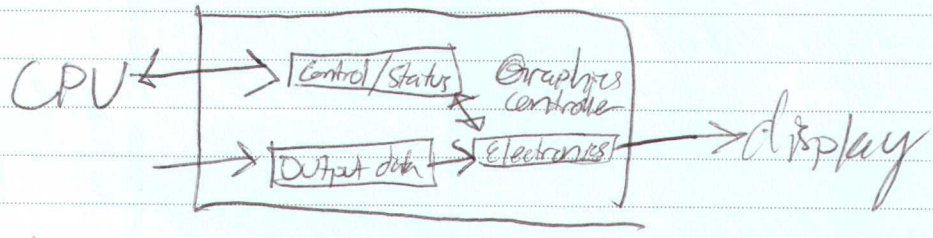
I/O:

Behavior

data rate

- CPU tells device where to go
- CPU checks whether task is done

Controller * (Control/Status Registers) + ( )



Device electronics

- performs actual operation
  ie - pixels on a screen

Programming Interface

How are devices identified?

- Memory Mapped VS. Special Instructions

How is it timed / transferred managed?

- asynchronous VS. synchronous

Who controls transfer?

CPU (polling) VS device (interrupt)

11/17/21

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

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

Memory-Mapped
- assign a memory address to each device reg.
- use data movement instructions (LD/ST)

Bandwidth: in given time how many bits can a machine get back

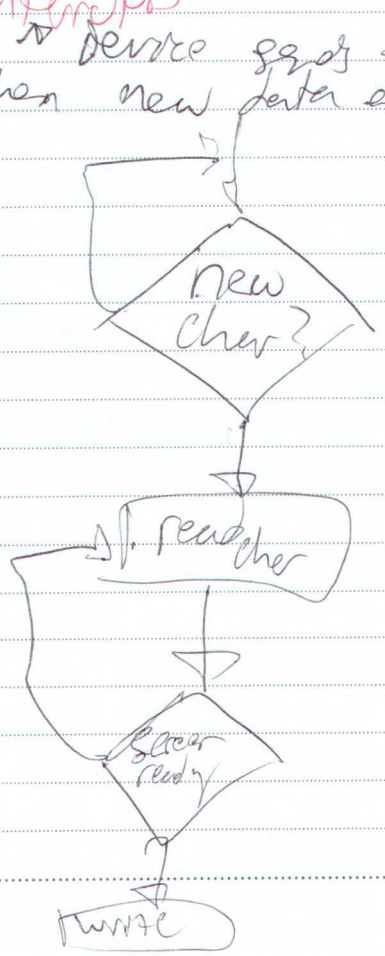
Synchronous:

Asynchronous:

KBSR: .FILL xFE00

Polling
→ CPU keeps checking, are we there yet?

Interrupt
→ device sends signal when new data arrives



2.3 - TRAP Mechanism

1. Use of service routines

2 - table of service routine addresses
- stored in memory
- called System Control Block

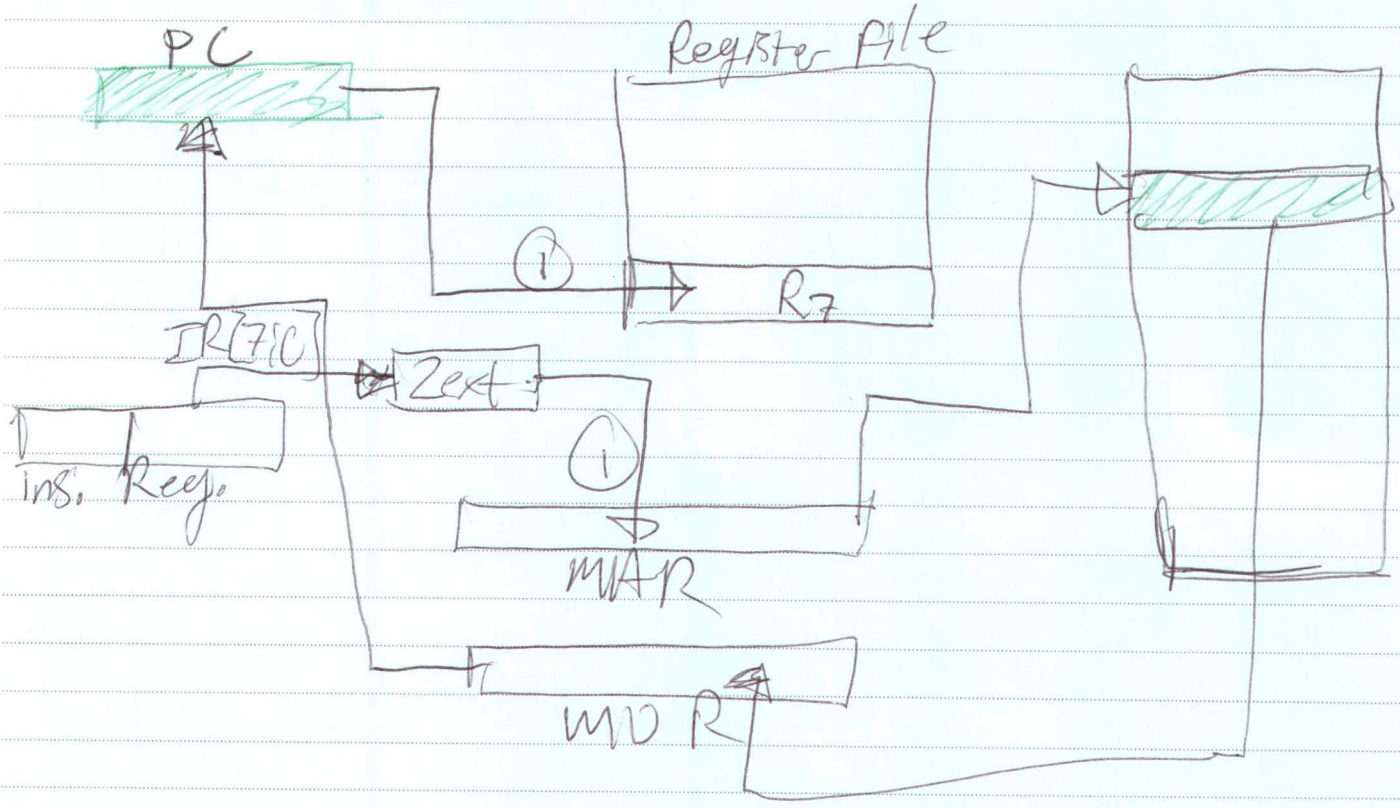
3 - TRAP instruction
- Used by program to transfer control to OS
- 8-bit trap vector names one of the 256 service routines

4 - a linkage back to the user program
- want execution to resume immediately after the trap

TRAP | 1111 | 0000 | trapvect8 |

Trap vector - - -

3.33



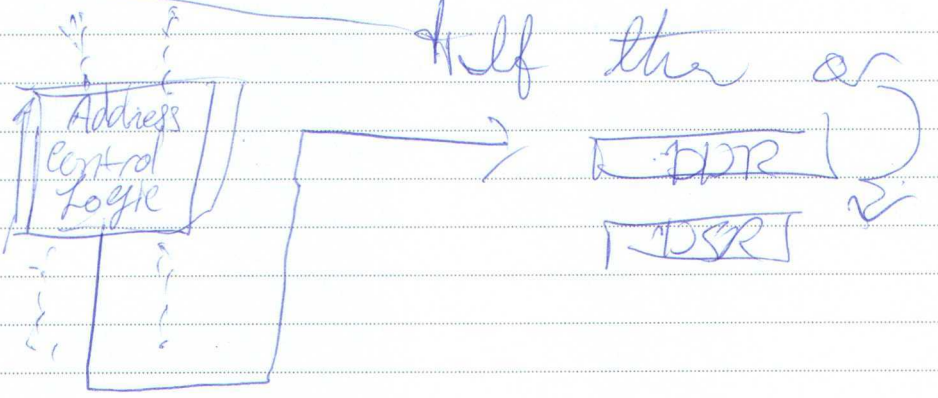
TOL Memory mapped I/O, Status Register,

Chapter 8 - In/output & Change of Traps

12 | 1 | 21

xFE00	KBSR	Keyboard Status Reg.
xFE02	KBDR	Keyboard Data Reg.
xFE04	DSR	Display Status Reg.
xFE06	DDR	Display Data Reg.

Trap Service Routines are located at address after x00FF and before 0x3000



Interrupt - Driven I/O

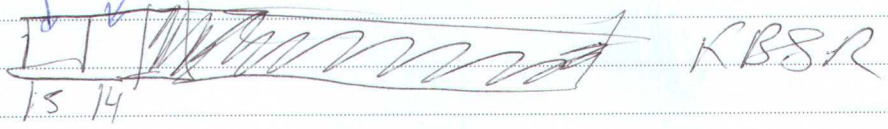
External device can

(1) force currently executing program to stop

(2) have the processor satisfy device's need

(3) Resume the stopped program

ready bit, interrupts enable bit.



* Overview | 12 8 21

- Boolean logic - representation
- One's, two's, floating,

Chapter 7,
Assembly - predict(i.e.) opcode, labels, #x literal
Directives - Pass argument to assembler
Assembler (wrangler) Linker - attach program stubs when linking codes

* Chapter 8 I/O
- Asynchronous v. Synchronous
- Reader polled v. Interrupt-driven
Trap Service Routine

Static v. Dynamic linking
Need a standard library
Creates binary decides the logical stent points of Std. library relative to code
adjusts the machine code