LC4 Instruction Overview

Introduction to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

How are you?

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Logistics

- HW03 Sequential Logic: **This Friday** 10/7 @ 11:59 pm
  - Written Homework, submitted to gradescope
  - **NO EXTENSIONS OVER 72 HOURS**
  - Should have everything you need
  - Practice in Recitations this week

- HW04 LC4 Programming: to be released this week
  - Programming assignment
  - May not have everything you need until Monday’s lecture

Lecture Outline

- **LC4 Review & shift instructions**
- Instructions as bits & HICONST
- Program Counter, JMP, BR, JSR
- LC4 misc. syntax

In-Person Lecture Policies

- I ask that you wear a mask in lecture

- If you are using your electronics (outside of polls), please sit in the back
  - Having electronics out make it a lot easier to distracted by random notifications
  - Easy for people sitting nearby & behind you to get distracted by your distractions

LC4 ASM vs C (Learning Example)

- Instead of operating on variables, we are operating on processor registers.
  - We have 8 of these: (R0, R1, R2 … R7)
  - (Program variables aren't just processor registers in reality, but we will treat them like that for now)

- Example comparing C cod to ASM:

C code

```
int R0 = 0;
int R1 = 12;

R0 = (R1 + 5)
R0 = R0 * R1;
```

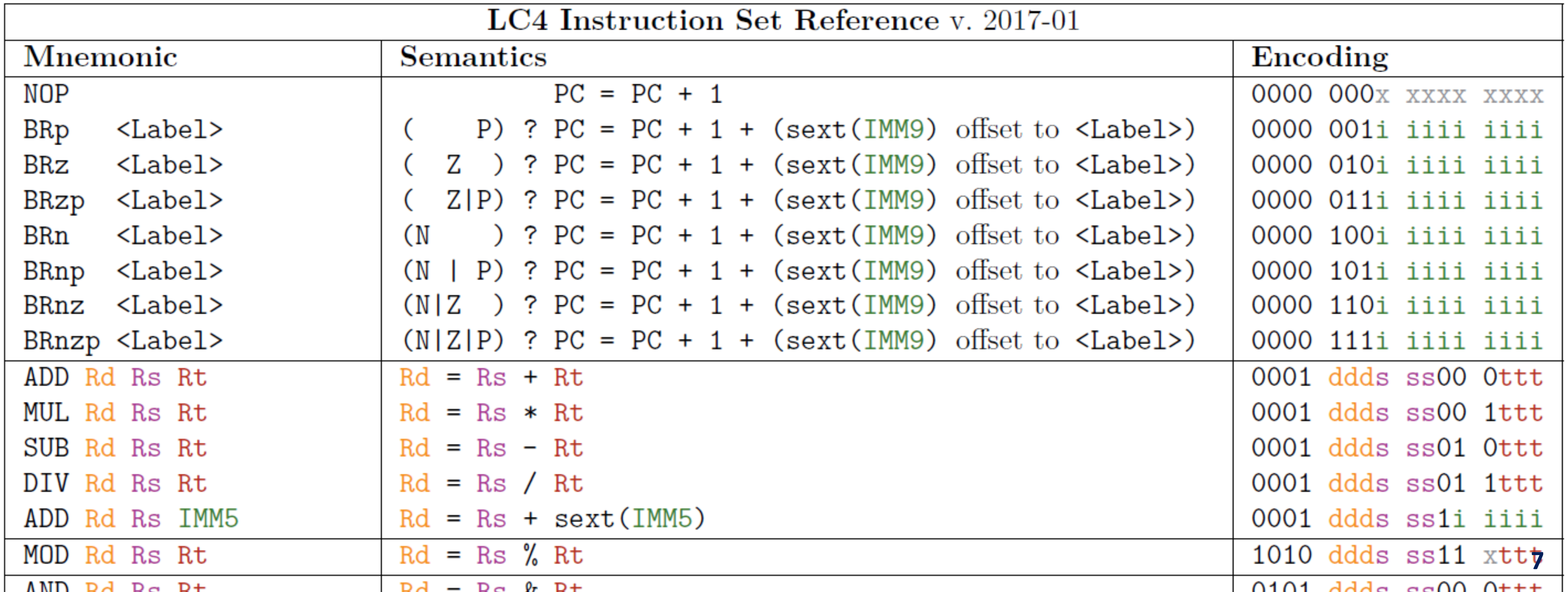
LC4

```
CONST R0, #0
CONST R1, #12

ADD R0, R1, #5
MUL R0, R1, R0
```

C doesn't translate into assembly this way; this is just a comparison for learning

LC4 "Cheat Sheet"

- Contains every LC4 instruction, its behaviour, and other information we will discuss later
  - On the website under "references"
  - HIGHLY recommend you print a copy
  - Will be provided on exams if needed



LC4 Instruction Set Reference v. 2017-01

Mnemonic	Semantics	Encoding
NOP	PC = PC + 1	0000 000x xxxx xxxx
BRp <Label>	(P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 001i iiii iiii
BRz <Label>	(Z) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 010i iiii iiii
BRzp <Label>	(Z|P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 011i iiii iiii
BRn <Label>	(N) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 100i iiii iiii
BRnp <Label>	(N | P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 101i iiii iiii
BRnz <Label>	(N|Z) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 110i iiii iiii
BRnzp <Label>	(N|Z|P) ? PC = PC + 1 + (sext(IMM9) offset to <Label>)	0000 111i iiii iiii
ADD Rd Rs Rt	Rd = Rs + Rt	0001 ddds ss00 0ttt
MUL Rd Rs Rt	Rd = Rs * Rt	0001 ddds ss00 1ttt
SUB Rd Rs Rt	Rd = Rs - Rt	0001 ddds ss01 0ttt
DIV Rd Rs Rt	Rd = Rs / Rt	0001 ddds ss01 1ttt
ADD Rd Rs IMM5	Rd = Rs + sext(IMM5)	0001 ddds ss1i iiii
MOD Rd Rs Rt	Rd = Rs % Rt	1010 ddds ss11 xttt
AND Rd Rs Rt	Rd = Rs & Rt	0101 ddds ss00 0ttt

All Arithmetic Instructions in LC4

- All arithmetic operations in LC4:

ADD Rd Rs Rt	Rd = Rs + Rt
MUL Rd Rs Rt	Rd = Rs * Rt
SUB Rd Rs Rt	Rd = Rs - Rt
DIV Rd Rs Rt	Rd = Rs / Rt
ADD Rd Rs IMM5	Rd = Rs + sext(IMM5)
MOD Rd Rs Rt	Rd = Rs % Rt

  - Note the order of registers matter for some operations
    - (DIV, SUB, MOD)
  - Note that DIV does integer division

Bitwise Instructions in LC4

- Bitwise operations in LC4:

```
AND Rd Rs Rt     | Rd = Rs & Rt
NOT Rd Rs        | Rd = ~Rs
OR  Rd Rs Rt     | Rd = Rs | Rt
XOR Rd Rs Rt     | Rd = Rs ^ Rt
AND Rd Rs IMM5   | Rd = Rs & sext(IMM5)
```

  - Very similar layout to arithmetic operations, just performing bitwise operations instead
  - Shifting also exists and will be discussed later

Poll Everywhere

pollev.com/tqm

- What is the final value of R0 after the following instructions are executed:

A. 0xFF

B. 0xF0

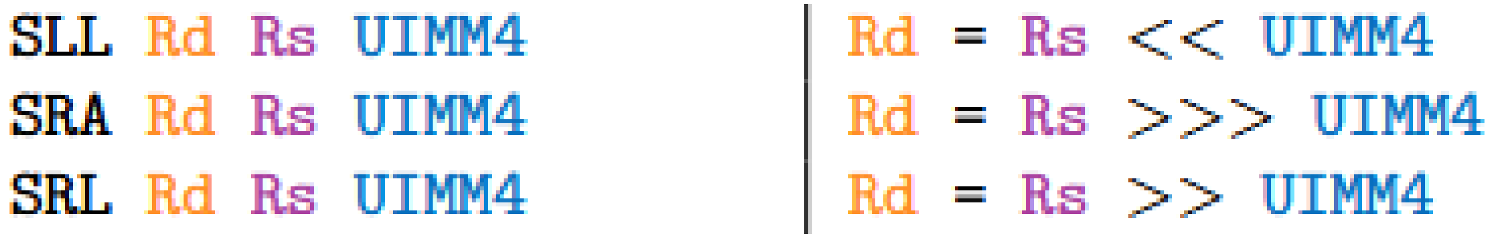
C. 0x01

D. 0xF1

E. I'm not sure

```
CONST R0, xFF
CONST R1, xF0
CONST R2, x01

AND R1, R2, R1
OR  R0, R0, R1
AND R0, R0, R2
```

Shift Instructions



```
SLL Rd Rs UIMM4   | Rd = Rs << UIMM4
SRA Rd Rs UIMM4   | Rd = Rs >>> UIMM4
SRL Rd Rs UIMM4   | Rd = Rs >> UIMM4
```

- Has all three shift types
  - SLL -> Shift Left
  - SRA -> Shift Right Arithmetic
  - SRL -> Shift Right Logical

Lecture Outline

- LC4 Review & shift instructions
- **Instructions as bits & HICONST**
- Program Counter, JMP, BR, JSR
- LC4 misc. syntax

Instruction Encodings

- Instructions are stored in memory over the lifetime of the program

- Each Instruction fills one memory location (16 bits)

- These 16 bits can be read to:
  - Identify the instruction
  - Identify the registers used in that instruction
  - Identify any integer constants used in that instruction

Encoding Example: Op-codes

- Many instructions are grouped into categories.
  - Arithmetic Instructions, Logical Instructions, Shift instructions…
- This group can be identified by the first 4-bits of the instructions called the **op-code**
  - If the group has more than one instruction, a **sub-op-code** stored elsewhere in the instruction is used to identify the instruction.
- Example:
- The op-code is the first four bits "0001"

$$\overbrace{\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}$$

Encoding Examples: ADD

16 bits broken into 4-bit chunks for easier reading

- **`ADD Rd, Rs, Rt`** encoding:

  `0001 ddds ss00 0ttt`

  - `0001` is the op-code, identifies this part of the arithmetic group
  - `000` is the sub-op-code which specifies that this is the **`ADD Rd, Rs, Rt`** instruction within the arithmetic group
  - **`ddd`**, **`sss`**, and **`ttt`** specify the corresponding register
    - Since there are only 8 registers, only 3 bits needed to specify a register ($8 = 2^3$)

- All instruction encoding formats are on the reference sheet ☺

Decoding Example:

- Consider the following 16 bits:

0	0	0	1	0	0	0	1	1	1	0	0	0	1	0	1

- Decoding steps:
  - Identify group by reading the op-code
  - Identify the instruction from sub-op-code if needed
  - Use the encoding format to decode the instruction

- `ADD Rd, Rs, Rt` encoding:

```
0001 ddds ss00 0ttt
```

Poll Everywhere

pollev.com/tqm

- What instruction does this 16-bit value represent?
  - Bonus, what registers and/or register constants are being used?

1	0	1	0	1	0	0	0	1	1	0	0	0	1	1	1

1	0	1	0

1	0	0	0

1	1	0	0

0	1	1	1

A. MOD

B. SLL

C. SRA

D. SRL

E. I'm not sure

```
MOD Rd Rs Rt       1010 ddds ss11 xttt
SLL Rd Rs UIMM4    1010 ddds ss00 uuuu
SRA Rd Rs UIMM4    1010 ddds ss01 uuuu
SRL Rd Rs UIMM4    1010 ddds ss10 uuuu
```

CONST Limitations

- Remember when I introduced CONST?
- **`CONST Rd, IMM9`**
  - Action: **`Rd = SEXT(IMM9)`**
  - Store an integer constant in the specified register
  - IMM9 = 9-bit 2C integer immediate
  - SEXT stands for **S**ign **Ext**ension.
  - A register is 16 bits, but the value we are storing is only 9 bits

- What if we wanted to set Rd to be a number that can't be represented in 9-bit 2C?

```
1001 dddi iiii iiii
```

Can only fit a 9-bit pattern in the instruction encoding ☹

CONST & HICONST

- If we wanted to set a register to value that can't be expressed in 9-bit 2C, we need to use CONST and then HICONST

- **`HICONST Rd, UIMM8`**
  - Action: **`Rd = (Rd & 0xFF) | (UIMM8 << 8)`**
  - Sets the upper 8 bits of **`Rd`** to be the specified 8-bit pattern.
  - Keeps the lower 8 bits of **`Rd`** the same.

Lecture Outline

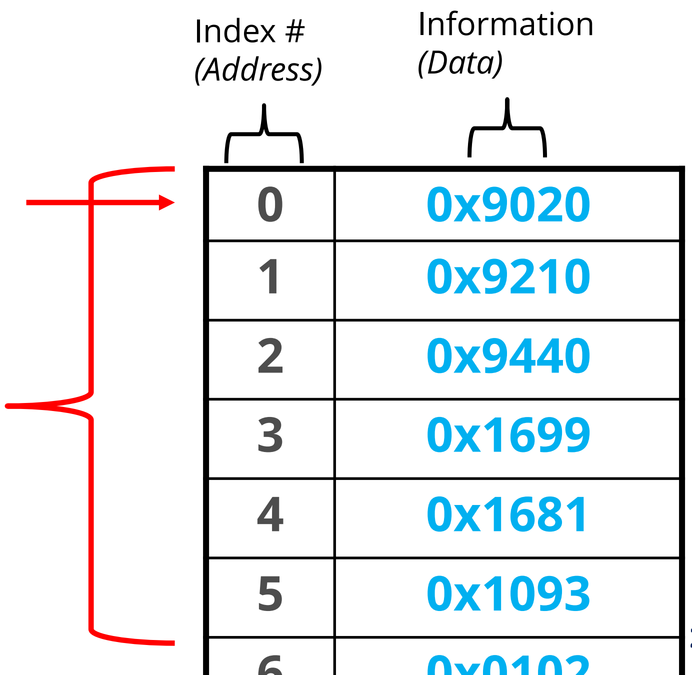
- LC4 Review & shift instructions
- Instructions as bits & HICONST
- **Program Counter, JMP, BR, JSR**
- LC4 misc. syntax

Code in Memory

- An instruction fits in 1 memory location (16 bits)

- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory

```
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
```

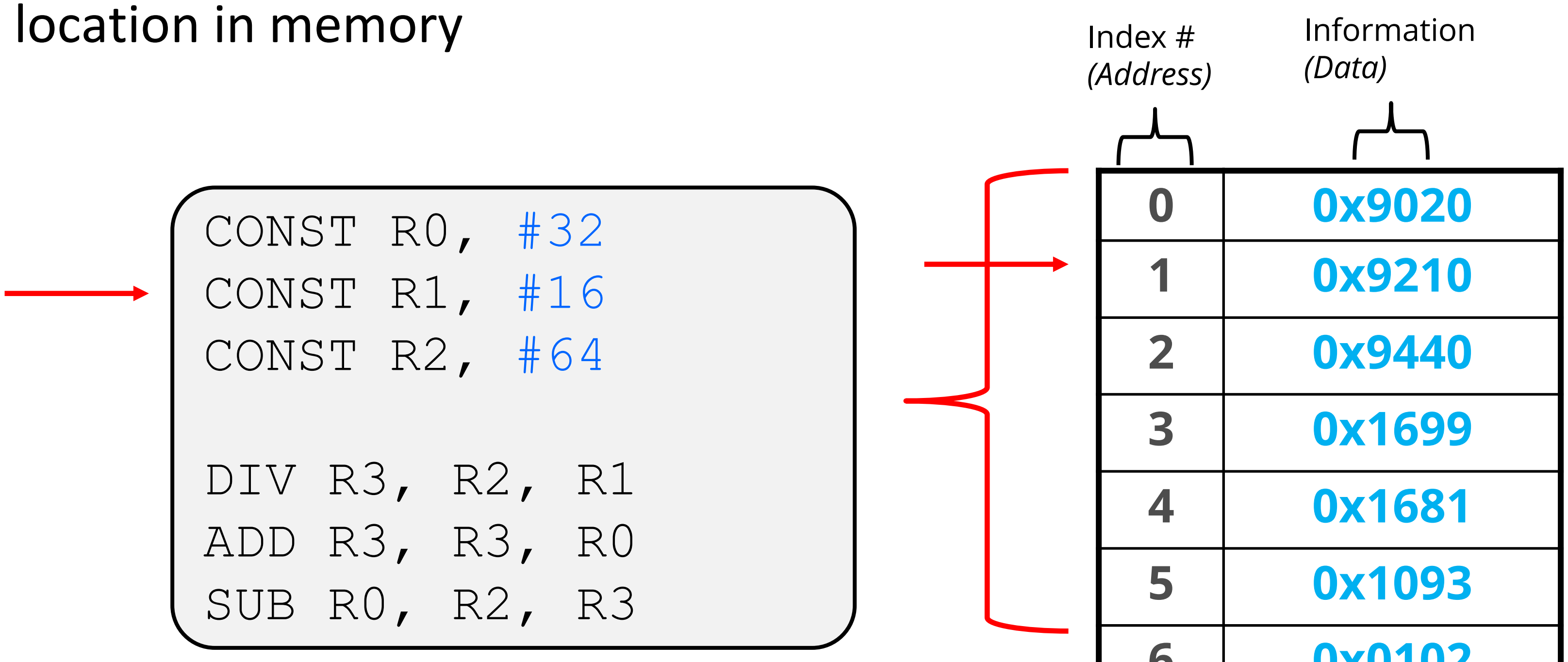

Index # *(Address)*
Information *(Data)*

Index # *(Address)*	Information *(Data)*
0	0x9020
1	0x9210
2	0x9440
3	0x1699
4	0x1681
5	0x1093
6	0x0102

Code in Memory

- An instruction fits in 1 memory location (16 bits)
- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory

```
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
```



Code in Memory

- An instruction fits in 1 memory location (16 bits)
- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory

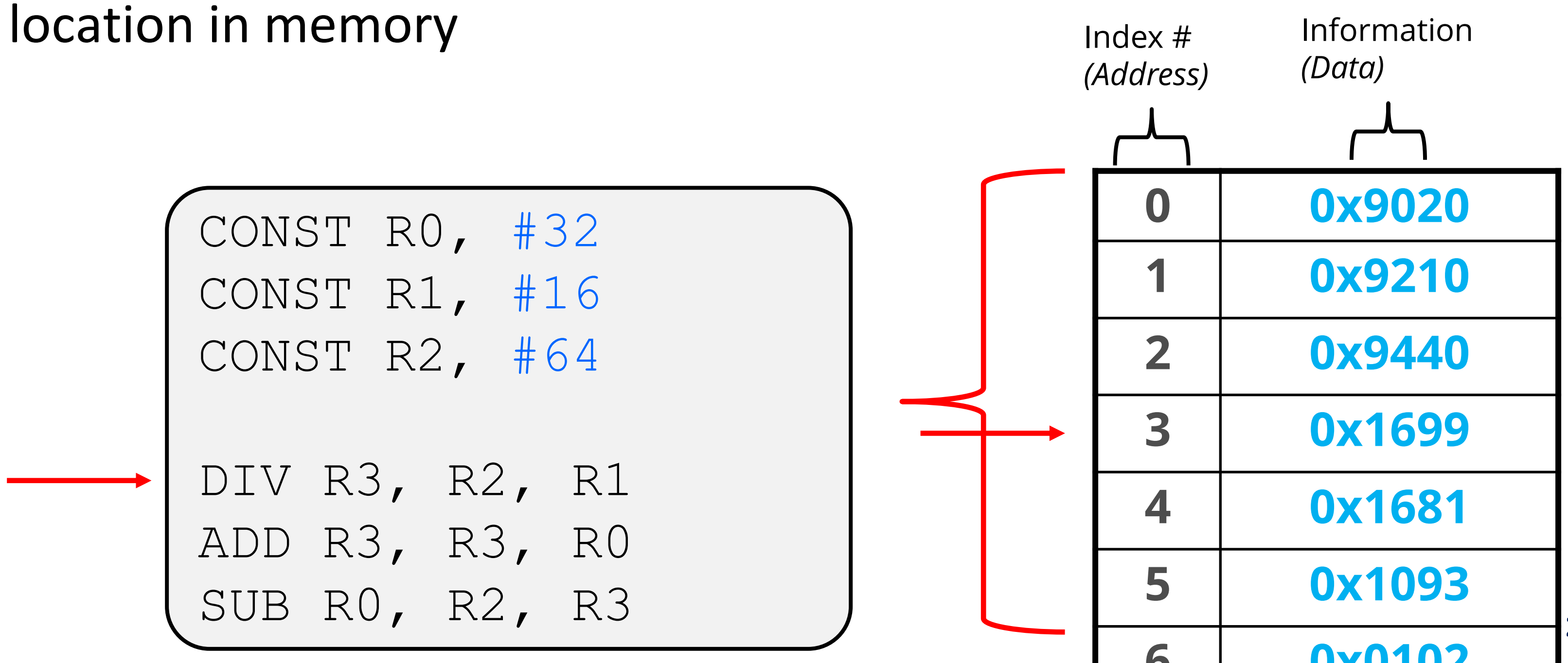
```
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
```

Index # *(Address)*	Information *(Data)*
0	**0x9020**
1	**0x9210**
2	**0x9440**
3	**0x1699**
4	**0x1681**
5	**0x1093**
6	**0x0102**

Code in Memory

- An instruction fits in 1 memory location (16 bits)

- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory



```
CONST R0, #32
CONST R1, #16
CONST R2, #64

DIV R3, R2, R1
ADD R3, R3, R0
SUB R0, R2, R3
```

Index # *(Address)*	Information *(Data)*
0	0x9020
1	0x9210
2	0x9440
3	0x1699
4	0x1681
5	0x1093
6	0x0102

Program Counter

- The **Program Counter** (PC) is a special register that keeps track of the address of the next instruction to execute

- Implicitly, every instruction we have covered so far also increments the PC

  - ADD doesn’t just perform addition, but also moves on to the next instruction to execute (the instruction after it)

Jump: JMP & JMPR

- **`JMP IMM11`**
  - Action: **`PC = PC + SEXT(IMM11) + 1`**
  - **`JMP`** == "Jump"
  - Modifies the **`PC`** by the specified amount + 1
  - IMM11 = 11-bit 2C integer immediate
  - SEXT stands for <u>**S**</u>ign <u>**Ext**</u>ension.

- **`JMPR Rs`**
  - Action: **`PC = Rs`**
  - **`JMPR`** == "Jump Register"
  - Updates the **`PC`** to hold the 16-bit address stored in **`Rs`**

Poll Everywhere

pollev.com/tqm

- What is the final value of R1 after executing this code?

A. 5

B. 10

C. 3

D. 6

E. I’m not sure

0	CONST R0, #3
1	CONST R1, #2
2	ADD R1, R1, R0
3	JMP #1
4	ADD R1, R1, #4
5	ADD R1, R1, #1
6	…

Lecture Outline

- LC4 Review & other “basic” instructions
- Instructions as bits
- Program Counter & JMP
- Conditional jumps
- Subroutine Calls
- Penn Sim

NZP

- LC4 has 3 bits that are reserved to keep track of NZP
  - NZP == **N**egative **Z**ero **P**ositive

- Anytime a register is written to, NZP is updated to reflect whether the value written was Negative, Zero or Positive.
  - The value written is interpreted as a 16-bit 2C number

- **`CONST R4, #0`**
  - Sets NZP to be 010 (N = 0, Z = 1, P = 0)

- If **`ADD R2, R4, #1`** is executed afterwards
  - Sets NZP to be 001 (N = 0, Z = 0, P = 1)

Conditional Jumps

- JMP and JMPR are instructions that always jump when executed. We may not always want to jump though.

- We can instead jump based on the status of the NZP bits using **`Br`** instructions

Branch Instructions

NZP itself can only ever be 100, 010, or 001

Mnemonic	Semantics	Encoding
`NOP`	PC = PC + 1	`0000000xxxxxxxxx`
`BRp IMM9`	(P) ? PC = PC + 1 + sext(IMM9)	`0000001IIIIIIIII`
`BRz IMM9`	(Z) ? PC = PC + 1 + sext(IMM9)	`0000010IIIIIIIII`
`BRzp IMM9`	(Z|P) ? PC = PC + 1 + sext(IMM9)	`0000011IIIIIIIII`
`BRn IMM9`	(N) ? PC = PC + 1 + sext(IMM9)	`0000100IIIIIIIII`
`BRnp IMM9`	(N | P) ? PC = PC + 1 + sext(IMM9)	`0000101IIIIIIIII`
`BRnz IMM9`	(N|Z) ? PC = PC + 1 + sext(IMM9)	`0000110IIIIIIIII`
`BRnzp IMM9`	(N|Z|P) ? PC = PC + 1 + sext(IMM9)	`0000111IIIIIIIII`

- Each possible way to test NZP bits has a corresponding branch instruction
  - If NZP test combination is 000, the branch always fails and so is considered a “NOP” (No Operation) and only increments PC
  - If NZP test combination is 111, the branch is always taken

Poll Everywhere

pollev.com/tqm

- What is the final value of R2 after executing this code?

A. 5

B. 2

C. 3

D. 4

E. I'm not sure

0	`CONST R0, #5`
1	`CONST R1, #2`
2	`CONST R2, #0`
3	`ADD R2, R2, #1`
4	`SUB R0, R0, R1`
5	`BRp #-3`
6	. . .

Comparison Instructions

Mnemonic	Semantics	Encoding
`CMP Rs, Rt`	NZP = sign(Rs - Rt)	`0010sss00----ttt`
`CMPU Rs, Rt`	NZP = sign(uRs - uRt)	`0010sss01----ttt`
`CMPI Rs, IMM7`	NZP = sign(Rs - SEXT(IMM7))	`0010sss10IIIIIII`
`CMPIU Rs, UIMM7`	NZP = sign(uRs - SEXT(UIMM7))	`0010sss11IIIIIII`

- Instructions that only modifies NZP and increments PC
  - Useful for checking a value before a BR instruction
  - Different variants for signed and unsigned values
  - Variants with immediate useful for things like `i > 0`, `i == 1`, `j < 10` etc.

- Performs comparison differently from the **`SUB`** instruction, **`CMP`** instructions can handle overflow safely

JSR and JSRR

- **`JSR IMM11`**
  - Action: **`R7 = PC + 1,`**
    `PC = (PC & 0x8000) | (IMM11 << 4)`
  - "Jump Subroutine"
  - Stores **`PC + 1`** in **`R7`** before jumping so that after the subroutine, we can return to right after **`JSR`**

- **`JSRR Rs`**
  - Action: **`R7 = PC + 1, PC = Rs`**
  - "Jump Subroutine Register"

- We use these to implement function calls in higher level languages (More on this later in the semester)

TRAP & RTI

- Instructions for making System Calls & leaving/entering the Operating System

- More on these in ~2 weeks when we briefly talk about what an operating system is.

Lecture Outline

- LC4 Review & shift instructions
- Instructions as bits & HICONST
- Program Counter, JMP, BR, JSR
- **LC4 misc. syntax**

More LC4 Syntax

- Integer Immediates (CONST, HICONST, ADD, SLL, etc.) can be either in hexadecimal or in decimal form
  - Hexadecimal constants **`0xFF`** or **`xFF`**
  - Decimal constants: **`#240`**, **`#-240`**

- Comments
  - Comments in LC4 are preceded by a ;
- Example:

```
CONST R0, 0x20
CONST R1, x10
CONST R2, #64
; this is a comment

DIV R3, R2, R1  ; this is a comment too
ADD R3, R3, R0
```

LC4 Labels

- It can be cumbersome to calculate offsets for jumps.
- LC4 assembler allows us to put labels on memory. We can use labels for Jumps and Branch instructions to make our lives easier
  - A Label is just a “name” for a memory location. Like how we can refer to a memory location with an address.

- Example:

```
        CONST R0, #5
        CONST R1, #2
        CONST R2, #0

LOOP    ADD R2, R2, #1
        SUB R0, R0, R1
        BRp LOOP
END     JMP #-1
```

Labels

Assembler will calculate the offset to the specified label for us

Next Lecture:

- PennSim Demo
- Program Design in LC4
- Accessing Memory in LC4
- Pointers, Arrays, Strings