EECS 370 - Lecture 3

LC2K

Important Data

Favorite Holidays

1. Christmas
2. Thanksgiving
3. Chinese New Year
4. Halloween

Honorable mention: Burrito Eating Day

Announcements

- Lab 1 assignment due **Wednesday**
  - No attendance required or pre-lab quiz for lab 1

- Pre-lab quiz for lab 2 posted on **Canvas, due Thursday**
- All later pre-lab quizzes will be posted on Monday, covers through Tuesday lecture, due on Thursday

- P1 posted
  - First part due Thursday September 12th (1.5 weeks)
  - You’ll have everything you need after Thursday’s lecture

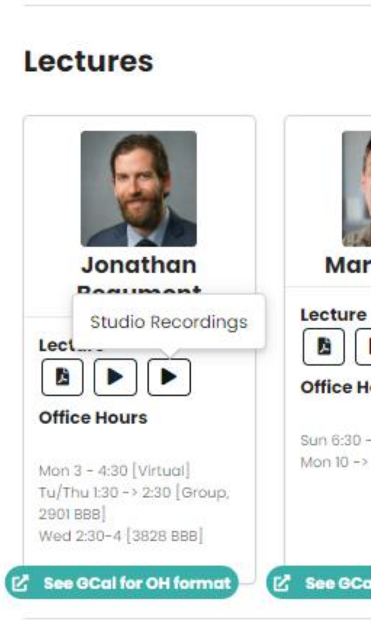
Instruction Set Architecture (ISA) Design Lectures

"People who are really serious about software should make their own hardware." — Alan Kay

- Lecture 2: ISA - storage types, binary and addressing modes
- **Lecture 3 : LC2K**
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- Lecture 6 : Converting C to assembly – functions
- Lecture 7 : Translation software; libraries, memory layout

Reminder: Studio Recordings

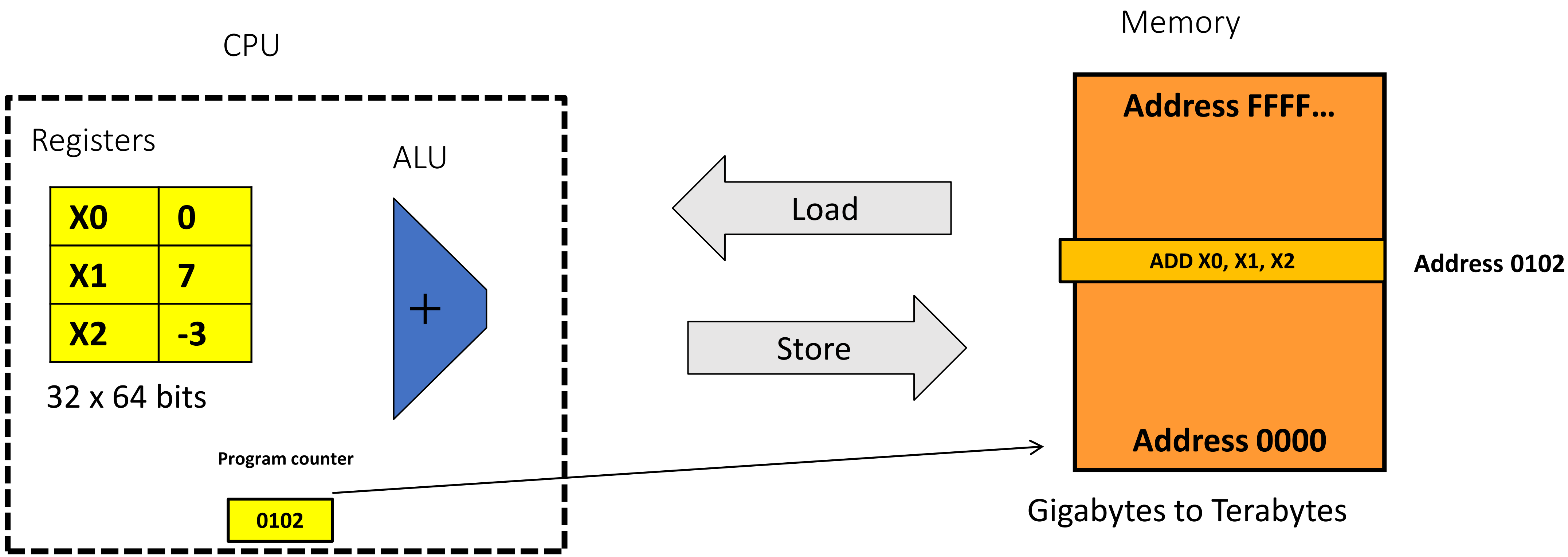
- If you're watching lectures asynchronously…
- I have studio recordings
  - Much better quality than lecture recordings



Reminder- System Organization

Let's execute this short program:

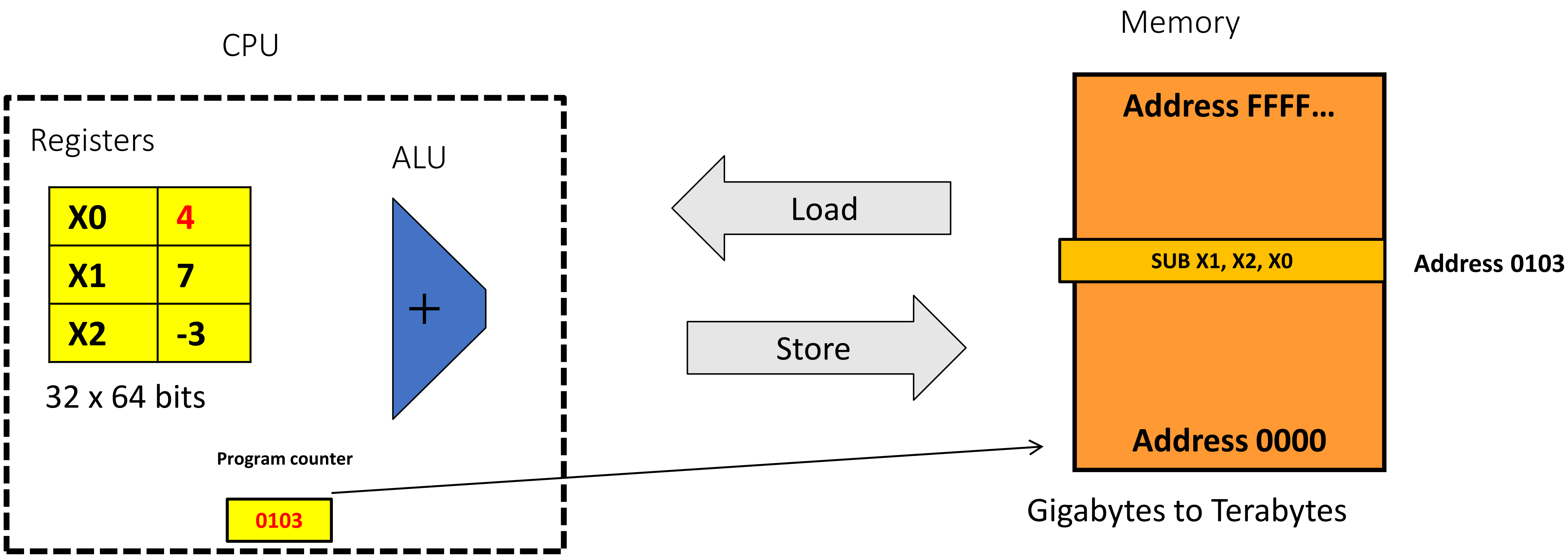
ADD X0, X1, X2
SUB X1, X2, X0



Reminder- System Organization

Let's execute this short program:

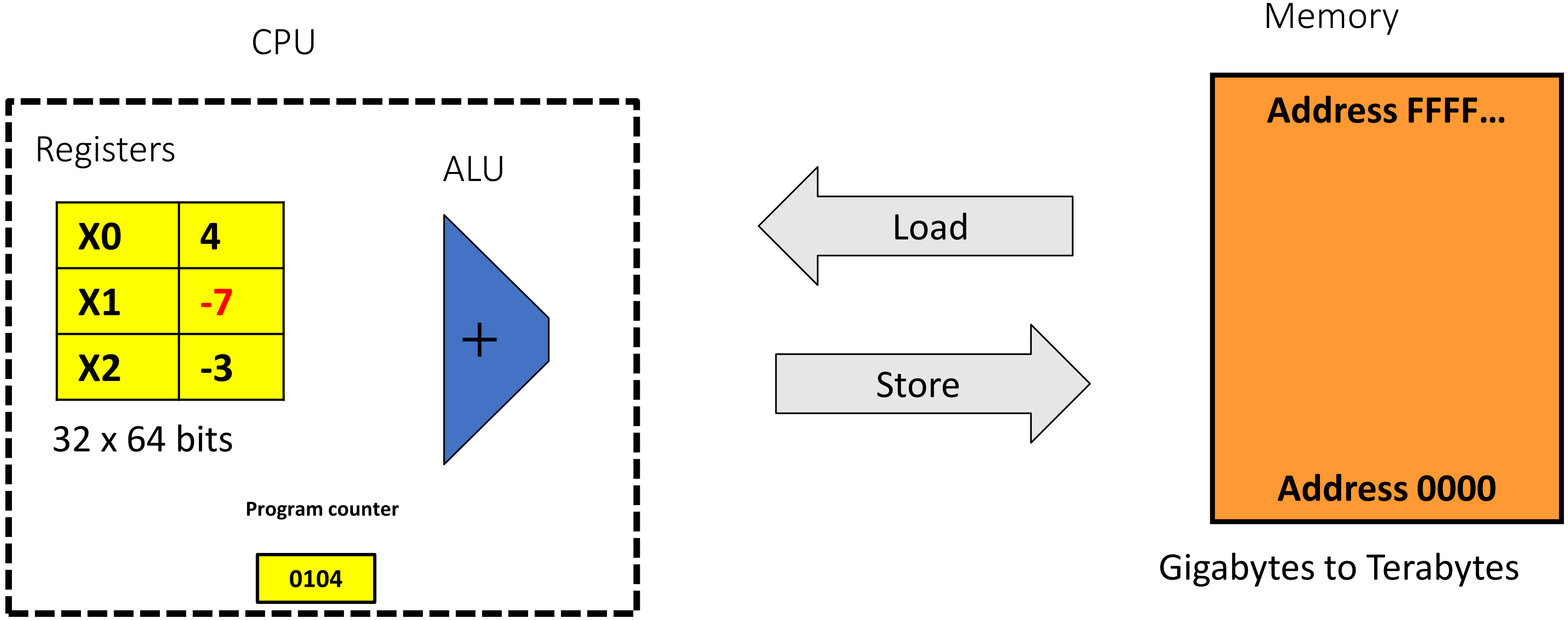
```
ADD X0, X1, X2
SUB X1, X2, X0
```



Reminder- System Organization

Let's execute this short program:

```
ADD X0, X1, X2
SUB X1, X2, X0
```



Different Data Types

- How does memory distinguish between different data types?
  - E.g. int, int *, char, float, double
- It doesn't! It's all just 0s and 1s!
- We'll see how to encode each of these later
- Exact length depends on architectures

How is Assembly Different from C/C++?

- No data types in assembly
- Everything is 0s and 1s: up to the programmer to interpret whether these bits should be interpreted as ints, bools, chars… or even instructions themselves!

```
char c = 'a';
c++; // c is now 'b'

// results in the same assembly as

int x = 97;
x++; // c is now 98

x = (int) c; // this instruction has no effect... why?
```

Minimum Datatype Sizzes

Type	Minimum size (bits)
char	8
int	16
long int	32
float	32
double	64

Representing Values in Hardware

- Unsigned integers represented as we've seen
- Chars are represented as ASCII values
  - e.g. 'a' -> 97, 'b' -> 98, '#' -> 35
- What about negative numbers?
- Fractional numbers?

Representing Negative Numbers

- There are many ways we could represent negative numbers
- Because it will eventually make our hardware simpler, the most common representation is 2's complement

No, not 2's *compliment!*

Two's Complement Representation

- Recall that 1101 in binary is 13 in decimal.

$$\underset{2^3}{1}\quad\underset{2^2}{1}\quad\underset{2^1}{0}\quad\underset{2^0}{1} = 8 + 4 + 1 = 13$$

- 2's complement numbers are very similar to unsigned binary numbers.
  - The only difference is that the first number is now negative.

$$\underset{-2^3}{1}\quad\underset{2^2}{1}\quad\underset{2^1}{0}\quad\underset{2^0}{1} = -8 + 4 + 1 = -3$$

Live Poll + Q&A: slido.com #eecs370

Fun with 2's Complement Numbers

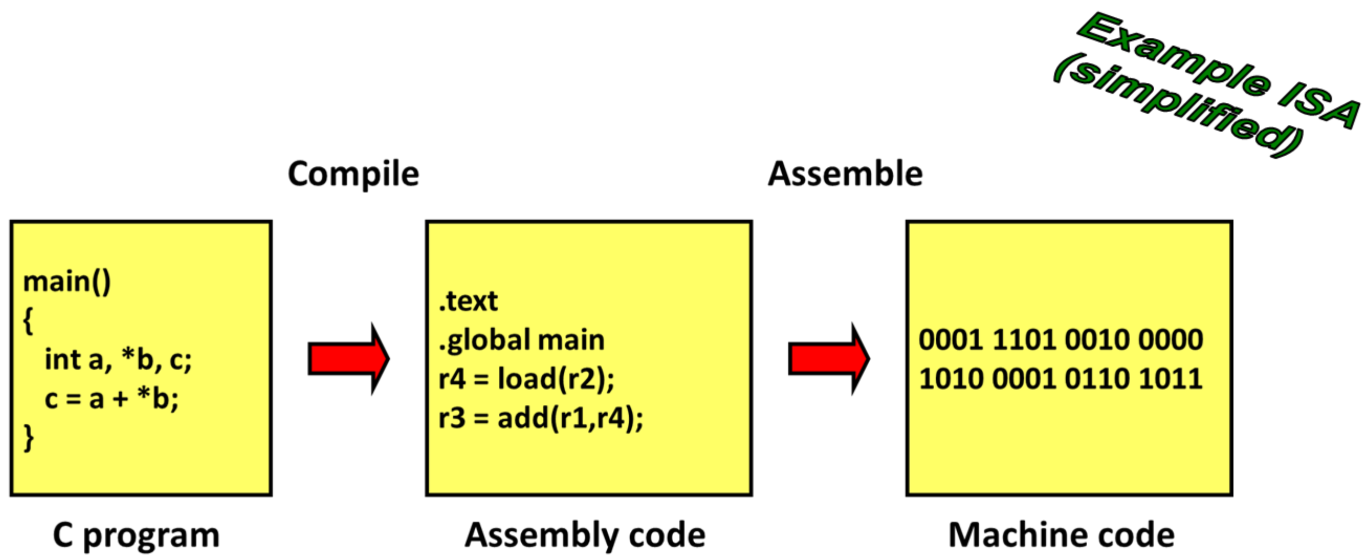
- What is the range of representation of a 4-bit 2’s complement number?
  - [-8, 7]   (corresponding to 1000 and 0111)
- What is the range of representation of an n-bit 2’s complement number?
  - $[-2^{(n-1)}, 2^{(n-1)} - 1]$
- Useful trick: You can negate a 2’s complement number by inverting all the bits and adding 1.
  - 5 is represented as **0101**
  - Negate each bit: **1010**
  - Add 1: **1011** = -8 + 2 + 1 = -5

What about fractional numbers?

- One idea: fixed point notation
  - Have some bits represent numbers before decimal point, some bits represent numbers after decimal point
- Better idea: floating point notation
  - Inspired by scientific notation (e.g. 1.3*10e-3)
  - Allows for larger range of numbers
  - We'll come back to this in a few lectures

Representing Instructions?

- Instructions, not just data, are stored in memory
- So, they must be expressible as numbers
- We'll look at how to encode instructions today



Agenda

- **LC2K Instruction Overview**
- Assembling LC2K into machine code
- Project 1a Overview
- Bonus Problems

LC2K Processor

LC2K ISA

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
  - register 0 always gives the value 0
- supports 65536 words of memory (addressable space)
- 8 instructions in the following common categories:
  - Arithmetic: add
  - Logical: nor
  - Data transfer: lw, sw
  - Conditional branch: beq
  - Unconditional branch (jump) and link: jalr
  - Other: halt, noop

These are enough instructions to express any computation*

****(that is not limited by memory size)***

LC2K Instruction Overview: add

```
add 1 2 3    // r3 = r1 + r2
```

- Pretty self-explanatory
- What if we want to do other arithmetic operations?
  - Subtract? Same as adding, but with a negated second operand
  - Negate? In 2's complement, bitwise-NOT followed by + 1
  - Multiply? You'll figure this out for P1m

LC2K Instruction Overview: nor

nor 1 2 3 // r3 = ~(r1 | r2)

- Treats each source operand as binary number
- Performs bitwise NOR for each pair of bits
  - E.g. if

r1 = 60 = **0b**0000_0000_0000_0000_0000_0000_0011_1100
r2 = 13 = **0b**0000_0000_0000_0000_0000_0000_0000_1101

then

r3 = 0b1111_1111_1111_1111_1111_1111_1100_0010

- What if we want other logical operations?
  - NOT? **nor** something with itself
  - AND? Can be done using De Morgan's Law (review if needed)

LC2K Instruction Overview: lw/sw

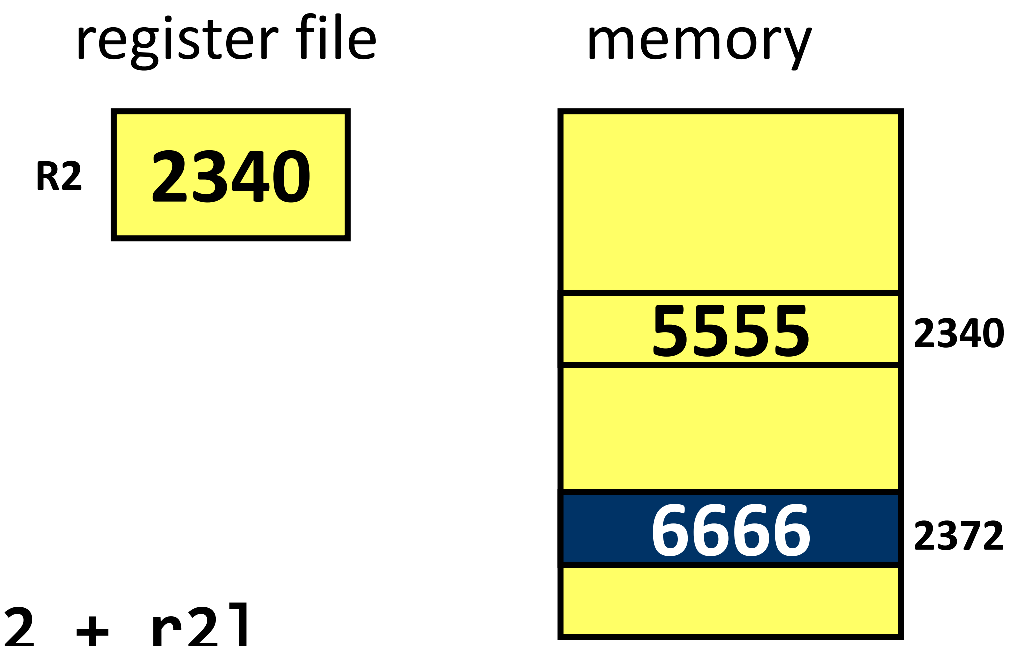
```c
// assume global variable
// is stored at address 1000
int GLOBAL;

int main() {
  GLOBAL = GLOBAL*2
}
```

```
lw  0 1 1000 // r1 = mem[1000+r0]
add 1 1 2    // r2 = 2*r1
sw  0 2 1000 // mem[1000+r0] = r2
```

- lw - "load word"
  - Loads a word (4 bytes) from a specified address into a register
- sw - "store word"
  - stores a word (4 bytes) from a register into a specified address
- Unlike add/nor, last operand here is **not** a register index
  - An **immediate** value: a number encoded directly in the instruction
- LC2K uses base+offset addressing
  - base register is first operand (if 0, then address = offset)

Non-Zero Displacement



- Consider this code:

```
struct My_Struct {
  int tot;
  //...
  int val;
};

My_Struct a;
//...
a.tot += a.val;
```

To load a.val...

`lw 2 1 32`

`// r1 = mem[32 + r2]`

(also need to load a.tot, add, then store... not shown here)

- If a register holds the starting address of "a"...
  - Then the specific values needed are at a slight **offset**
- **Base + Displacement**
  - reg value + immed

LC2K Instruction Overview: beq

```
beq 1 2 7 // if (reg1==reg2), PC=PC+1+7
```

- Remember: each line in assembly corresponds to a memory address
- "Program Counter" (PC) keeps track of address of current instruction
- Normally increments by 1
- "Branch if equal" (beq) allows us to change PC a different amount if 2 registers are equal
- Allows us to implement if/else statements, for/while loops
  - (example later)

LC2K Instruction Overview: the others

- jalr: used for function calls and returns
  - It's a bit complicated: we'll discuss later
- halt: ends the program
- noop:
  - "no operation"
  - Doesn't do anything
  - (We'll see later why this can be useful)

Note on Practical ISAs

- LC2K is made up for this class
- It's intended to be as simple as possible
  - Makes most of our projects less tedious
  - However, corresponding assembly code is **bloated**
- Practical ISAs will add many more instructions
  - Often hundreds, maybe thousands
  - Although functionally redundant, programs will be faster and easier to write

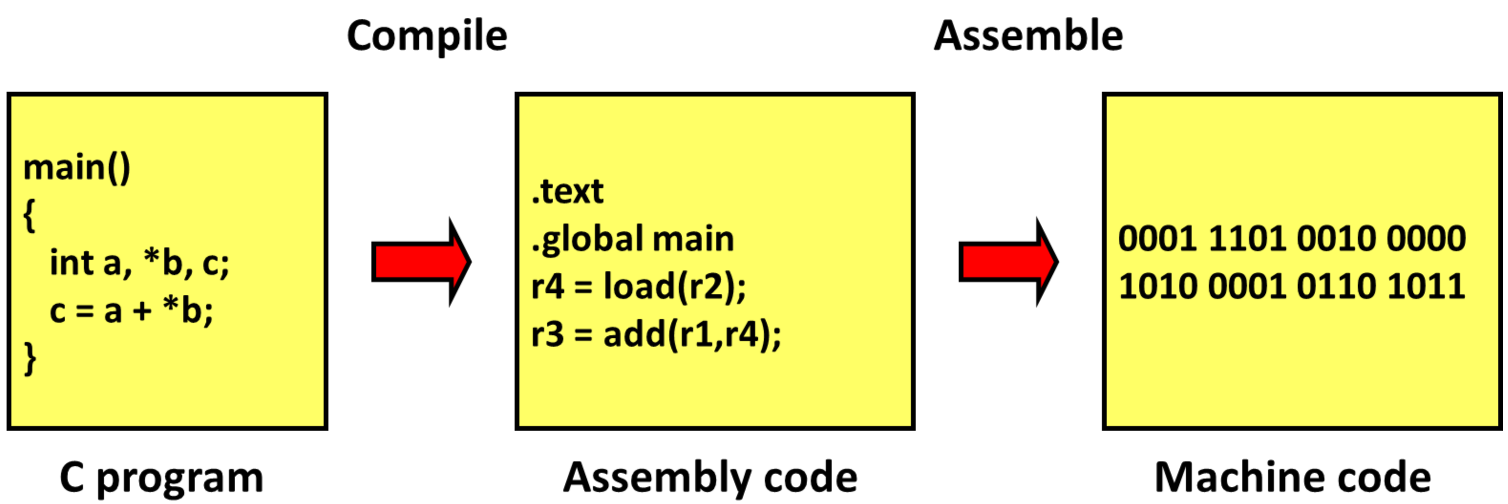
Agenda

- LC2K Instruction Overview
- **Assembling LC2K into machine code**
- Project 1a Overview
- Bonus Problems

Instruction Encoding

- Remember: computer doesn't understand text
  - Only understands 0s and 1s
- In order to execute our programs, assembly instructions must be converted into numbers
  - Corresponding numbers called the **machine code**
- Let's see how this is done with LC2K instructions

Example ISA (simplified)



LC2K ISA

Instruction Encoding

- Instruction set architecture defines the mapping of assembly instructions to machine code

add 1 2 3 // r3 = r1 + r2

<u>Cookies or brownies?</u>

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

<u>Poll:</u> If we wanted to extend the operation code (opcode) to use all the leading unused bits, how many operations could be supported?

LC2K ISA

Instruction Formats

- Tells you which bit positions mean what
- R (register) type instructions (add, nor)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

- I (immediate) type instructions (lw, sw, beq)

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

LC2K ISA

Instruction Formats

- J-type instructions (jalr)

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	*unused*

- O type instructions (halt, noop)

31-25	24-22	21-0
unused	**opcode**	*unused*

Bit Encodings

LC2K ISA

- Most significant bits (besides unused 31-25) consist of the operation code or **opcode**
  - Indicates what type of operation
  - LC2K has 8 instructions, so we need $\log_2 8 = 3$ bits for the opcode
- Opcode encodings
  - add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)
- Register values
  - 8 registers, so $\log_2 8 = 3$ bits for each register index
  - Just encode the register number (r2 = 010)
- Immediate values
  - Just encode the values in **2's complement format**

Fun with 2's Complement Numbers

- What is the range of representation of a 4-bit 2’s complement number?
  - [-8, 7]   (corresponding to 1000 and 0111)
- What is the range of representation of an n-bit 2’s complement number?
  - $[-2^{(n-1)}, 2^{(n-1)} - 1]$
- Useful trick: You can negate a 2’s complement number by inverting all the bits and adding 1.
  - 5 is represented as **`0101`**
  - Negate each bit: **`1010`**
  - Add 1: **`1011`** = -8 + 2 + 1 = -5

LC2K ISA

Example Encoding - nor

- nor 3 4 7 (r7 = r3 nor r4)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

LC2K ISA

Example Encoding - nor

- nor 3 4 7 (r7 = r3 nor r4)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**
0000000	001	011	100	0000000000000	111

Convert to Hex → 0x005C0007

Convert to Dec → 6029319

LC2K ISA

Example Encoding - lw

- lw 5 2 -8 (r2 = mem[r5 + -8])

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

LC2K ISA

Example Encoding - lw

- lw 5 2 -8 (r2 = mem[r5 + -8])

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**
0000000	010	101	010	1111111111111000

Convert to Hex → 0x00AAFFF8

Convert to Dec → 11206648

Note that we "bit-extend" 1 for negative numbers

Another way to think about the assembler

- Each line of assembly code corresponds to a number
  - “add 0 0 0” is just 0.
  - “lw 5 2 -8” is 11206648
- We only write in assembly because it’s easier to read and write

.fill

`int GLOBAL = 7;`

How can we hardcode this 7 in memory?

- I also might want a number, to be, well, a number.
  - Maybe I want the number 7 as a data element I can use.
- .fill tells the assembler to put a number instead of an instruction
- The syntax is just ".fill 7".
- Question:
  - What do ".fill 7" and "add 0 0 7" have in common?

.fill with lw / sw

- We most often use .fill along with lw or sw
- Remember: every line in an assembly program corresponds to an address in memory
  - When an instruction is to be executed, that address is sent to memory
- `".fill 71"` is address 2, meaning mem[2]=71
- `"lw 0 1 2"` loads the contents of mem[2] into register 1

```
lw    0   1   2
halt
.fill 71
```

```
8454146
25165824
71
```

.fill

- **.fill is NOT an instruction**

- It does not have a corresponding opcode

- It should be used to initialize data in your program
  - If your PC ever points to it, something has probably gone wrong

- But if the PC **DOES** point to it, it will treat it as whatever type of instruction encodes to that number

IMPORTANT!

Labels in LC2K

- The code on the right is awkward
  - Need to count lines to see what it's doing
- **Labels** make code easier to read/write
- Label **definition**: listed to the **left** of the opcode
  - Can be defined on any line (only once)
- Label **use**: replaces offset value in lw/sw/beq instructions (any number)
- For lw/sw, assembler will replace label use with the line number of definition
  - In this example, data is on line 2

```
      lw    0  1  2
      halt
      .fill 71
```

```
      lw    0  1  data
      halt
data  .fill 71
```

Use of data **label**

Definition of data **label**

Labels in LC2K - beq

- Labels with beq indicate **where** we should branch
- Assembler's job is a little tricker
  - Doesn't just replace it with line number
  - Remember: target address is PC+1+offset

```
        beq   3   4   1
        add   1   2   3
        halt
```

```
        beq   3   4   end
        add   1   2   3
end     halt
```

If reg3==reg4, skip to this line

…else do this line first

Exercise

```
// this is the assembly for:
while(x != y) {
  x *= 2;
}
```

- What are the values of the labels here?

```
loop  beq   3   4   end
      add   3   3   3
      beq   0   0   loop
end   halt
```

Poll: **What are the labels replaced with?**

Agenda

- LC2K Instruction Overview
- Assembling LC2K into machine code
- **Project 1a Overview**
- Bonus Problems

Programming Assignment #1

- Write an assembler to convert input (assembly language program) to output (machine code version of program)
  - “1a”
- Write a behavioral simulator to run the machine code version of the program (printing the contents of the registers and memory after each instruction executes
  - “1s”
- Write an efficient LC2K assembly language program to multiply two numbers
  - “1m”

Programming Assignment #1

- Where to start…
  - **<u>Write some test cases</u>** to check your code
    - Program 1: halt
    - Program 2: noop
    - halt
    - Program 3: add 1 1 1
    - halt
    - Program 4: nor 1 1 1
    - halt

Next Time

- The ARM ISA

Extra Problems

Agenda

- LC2K Instruction Overview
- Assembling LC2K into machine code
- Project 1a Overview
- **Bonus Problems**

LC2K ISA

Extra Problem 1

- Compute the encoding in Hex for:
  - add 3 7 3 (r3 = r3 + r7) (add = 000)
  - sw 1 5 67 (M[r1+67] = r5) (sw = 011)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

LC2K ISA

Extra Problem 1

- Compute the encoding in Hex for:
  - add 3 7 3 (r3 = r3 + r7) (add = 000)
  - sw 1 5 67 (M[r1+67] = r5) (sw = 011)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**
0000000	000	011	111	000…000	011

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**
0000000	011	001	101	0000000001000011

Extra problem 2

```
loop   lw    0   1   one
       add   1   1   1
       sw    0   1   one
       halt
one    .fill 1
```

<u>Poll:</u> What's the first line in binary?

- What does that program do?
- Be aware that a beq uses PC-relative addressing.
  - Be sure to carefully read the example in project 1.