# 3. Addressing modes and the LC2K ISA

**EECS 370 – Introduction to Computer Organization – Winter 2023**

**EECS Department**
**University of Michigan in Ann Arbor, USA**

# REVIEW AND REMINDERS

# Announcements—Reminders

- Project 1 assigned
  - Project 1.a due Thursday 1/26
  - Project 1.s and 1.m due Thursday 2/2
- Homework 1 assigned
  - Due Monday 1/23 on Gradescope
- Friday discussion is a setup clinic
  - Setting up your IDE etc.
  - No meetings on Monday (if you have discussion on Monday and need help with setup, come on Friday).

# Instruction Set Architecture (ISA) Design Lectures

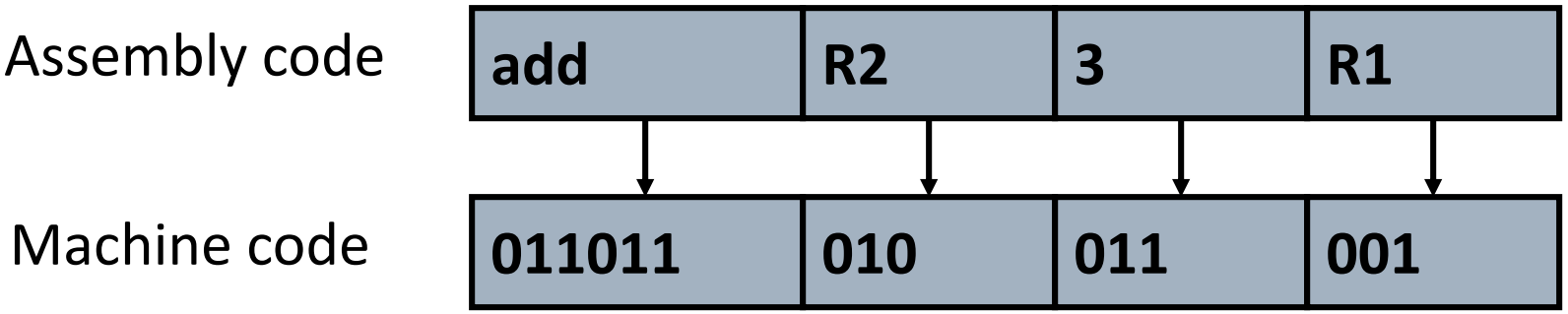
*"People who are really serious about software should make their own hardware." — Alan Kay*

- Lecture 2: ISA - storage types, binary
- **Lecture 3 : Addressing modes and 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

# Recap (Assembly/Machine Code)

- Computers store program instructions the same way they store data.

- Each instruction is encoded as a number
  - Opcode field: what instruction to perform.
  - Operand fields: what data to perform it on.



# Instruction Set Design – design space (1/2)

- What instructions should be included?
  - add, multiply, divide, sqrt [functions]
  - branch [flow control]
  - load/store [storage management]
- What storage locations?
  - How many registers?
  - How much memory?
  - Any other “architected” storage?
- How should instructions be formatted?
  - 0, 1, 2 or more operands?
  - Immediate operands

# Instruction Set Design – design space (2/2)

- How to encode instructions?
  - **RISC** (Reduced Instruction Set Computer): all instructions are same length (e.g. ARM, LC2K)
  - **CISC** (Complex Instruction Set Computer): instructions can vary in size (Digital Equipment's VAX, x86)
- What instructions can access memory?
  - For ARM and LC2K, only loads and stores can access memory (called a "**load-store architecture**")
  - Intel x86 is a "**register-memory architecture"**, that is, other instructions beyond ld/st can access memory

# Why study Instruction Set Design?

- Isn't there only one?
  - No, and even if there were, it would be too messy for a first course in computer architecture
- How often are new architectures created?
  - Embedded processors are designed all the time
  - Even the Intel x86 ISA changes (MMX, MMX2, SSE, SSE2, AVX, … )
- Will I ever get to (have to) design one?
  - Very possible…

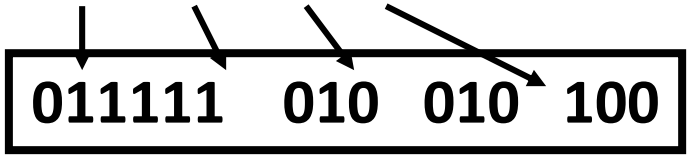
# Storage Architecture

1. Immediate Values
   - Specifying constants in instructions
2. Registers
   - Fast and small (and useful)
3. Memory
   - Big and complex (and useful)

# 1. Immediate Values

= small constant values placed in instructions

- They are stored in memory only because all instructions are in memory
- Useful for loading small constants
  - Example: ptr++; → addi R2, R2, #4




  - Very useful for branch instructions

    → target address is often immediate – in the instruction
- Size of the immediate is usually determined by how many bits are left in the instruction format.

# 2. Register Storage

- First came the **accumulator**, a single register architecture
  - Example: add #5
  - You don't need to specify which register when you only have one!
- Register File
  - Small array of **memory-like** storage
  - Register access is faster than memory because register file arrays are small and can be put right next to the functional units in the processor.
- Each register in the register file has a specific size
  - e.g. 32-bit registers
- Also called "**register addressing**"

# Example Architectures

- ARMv8—LEGv8 subset from P+H text book
  - 32 registers (X0 – X31)
  - 64 bits in each register
  - Some have special uses e.g. X31 is always 0—XZR
- Intel x86
  - 4 general purpose registers (eax, ebx, ecx, edx) 32 bits
    - You can treat them as two 8 or one 16-bits as well (ah, al, and ax)
  - Special registers: 3 pointer registers (si,di,ip), 4 segment (cs,ds,ss,es),

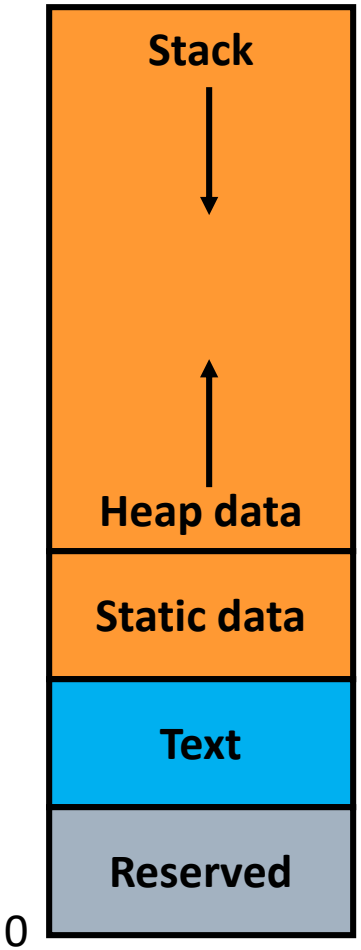
    2 stack (sp, bp), status register (flags)
- LC2K (the architecture you will be simulating)
  - 8 registers, 32 bits each

# 3. Memory Storage

- Large array of storage accessed using memory addresses
  - A machine with a 32 bit address can reference memory locations 0 to $2^{32}$-1 (or 4,294,967,295).
  - A machine with a 64 bit address can reference memory locations 0 to $2^{64}$-1 (or 18,446,744,073,709,551,615—18 exa-locations)

- Lots of different ways to calculate the address.

# Memory architecture: The ARM (Linux) Memory Image




Activation records: local variables, parameters, etc.

Dynamically allocated data—new or malloc()

Global data and static local data

Machine code instructions (and some constants)

Reserved for operating system

# ADRESSING MODES

# Addressing Modes

- Direct addressing
- Register indirect
- Base + displacement
- PC-relative

# Direct Addressing

- Like register addressing
  - Specify address as immediate constant
  - load r1, M[1500] ; r1 ← contents of location 1500
  - jump M[3000] ; jump to address 3000

**Not practical in modern ISAs…**

**if we have 32 bit instructions and 32 bit addresses, the entire instruction is the address!**

- Useful for addressing locations that don't change during execution
  - Branch target addresses
  - Global/static variable locations

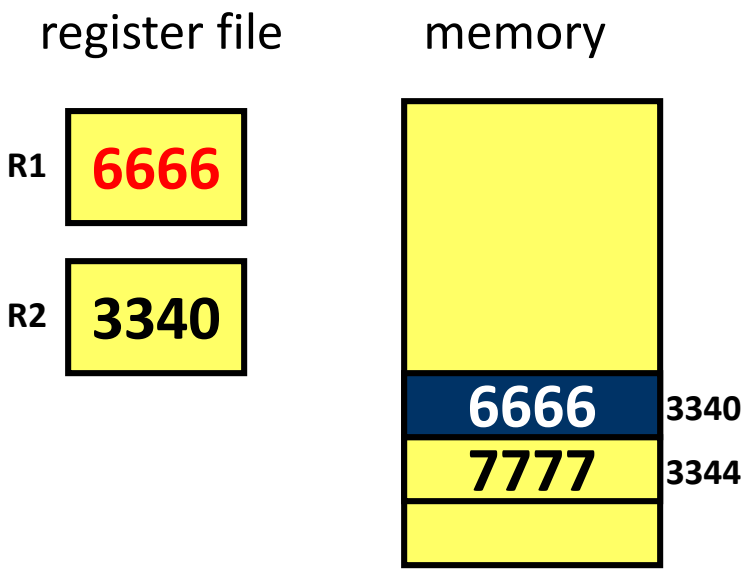
# Register indirect

- Specify which register has the reference address
  - Very useful for pointers

load r1, M[ r2 ]

*add r2, r2, #4*

*load r1, M[ r2 ]*



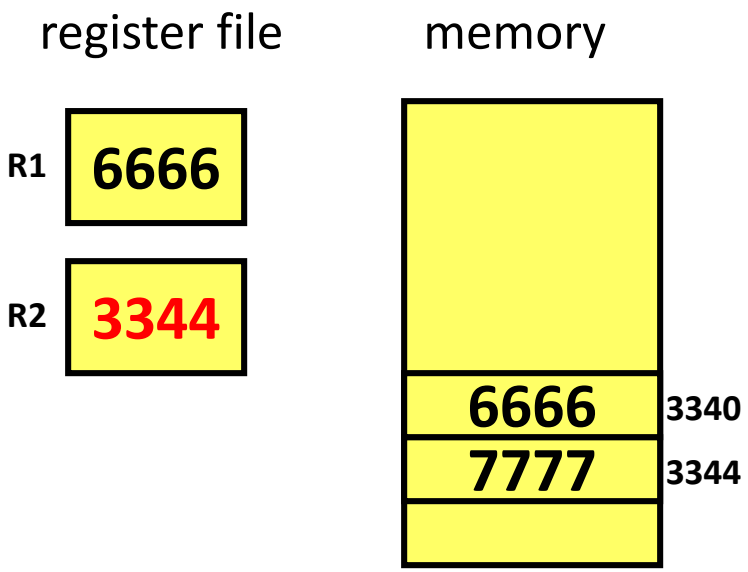
# Register indirect

- Specify which register has the reference address
  - Very useful for pointers

*load r1, M[ r2 ]*

add r2, r2, #4

*load r1, M[ r2 ]*



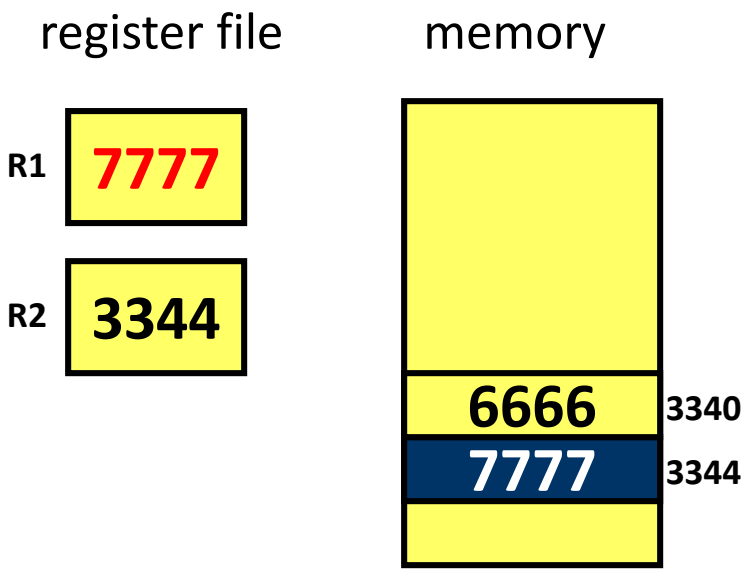
# Register indirect

- Specify which register has the reference address
  - Very useful for pointers

*load r1, M[ r2 ]*

*add r2, r2, #4*

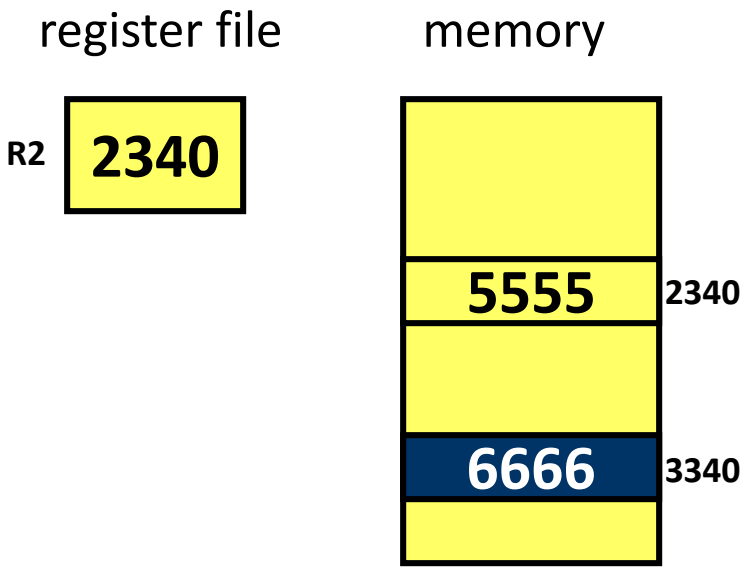
load r1, M[ r2 ]



# Base + Displacement

- Most common addressing mode today
- Compute address as: reg value + immed
- load r1, M[ r2 + 1000]
- Good for accessing class objects/structures. Why?

**a.tot += a.val;**

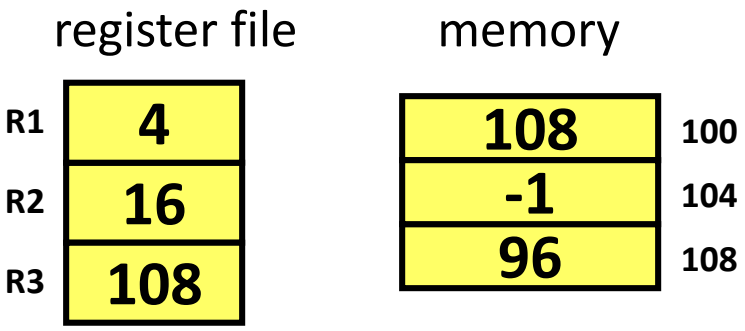



**Very general, most common addressing mode today!**

# Class Problem

**What are the contents of register/memory after executing the following instructions?**

r2 = load M[r3+0]
r3 = load M[r2+4]
r3=r3+r1
store M[r2+8], r3



# PC-relative addressing

- Variant on base + displacement
- PC register is base, longer displacement possible since PC is assumed implicitly
  - Used for branch instructions
    - jump [ - 8 ] ; jump back 2 instructions (32-bit instructions)

- Humans use labels and leave it to the assembler to determine the immediate value. Why?

# ISA Types

## Reduced Instruction Set Computing (RISC)

- Fewer, simpler instructions
- Encoding of instructions are usually the same size
- Simpler hardware
- Program is larger, more tedious to write by hand
- E.g. LC2K, RISC-V, ARM (kinda)
- More popular now

## Complex Instruction Set Computing (CISC)

- More, complex instructions
- Encoding of instructions are different sizes
- More complex hardware
- Short, expressive programs, easier to write by hand
- E.g. x86
- Less popular now

# LC2K ISA

# LC2K Processor

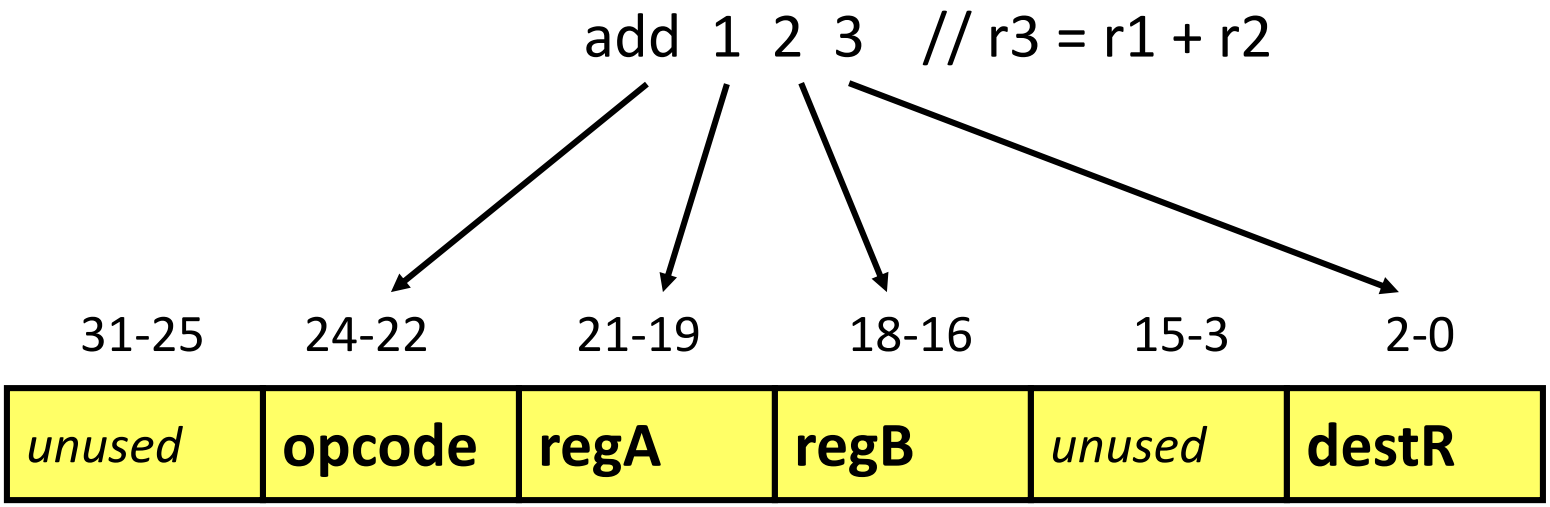
LC2K ISA

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- 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

# Instruction Encoding

LC2K ISA

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

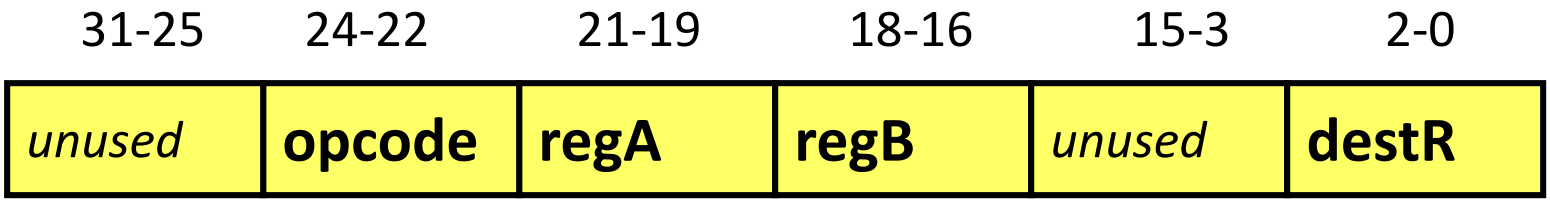


# Instruction Formats

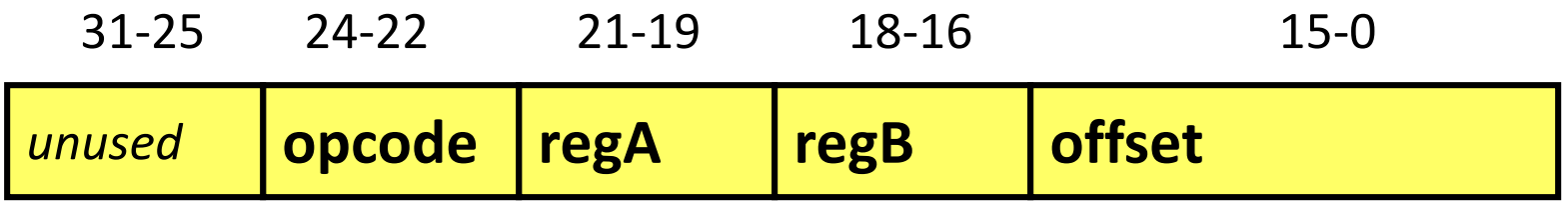
LC2K ISA

- Tells you which bit positions mean what

- R type instructions (add ‘000’, nor ‘001’)



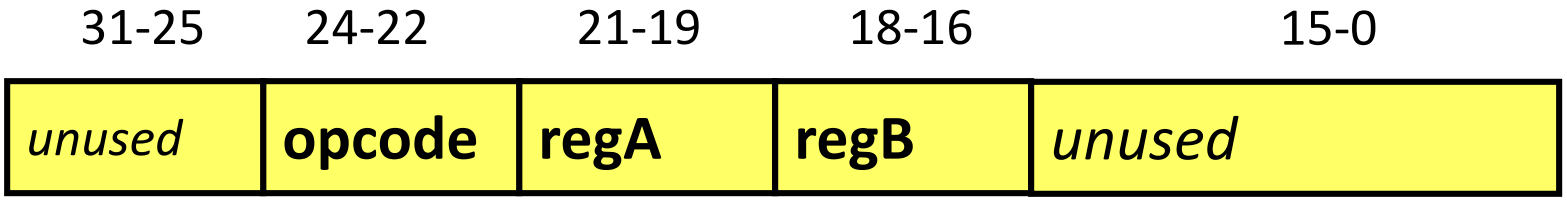

- I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)



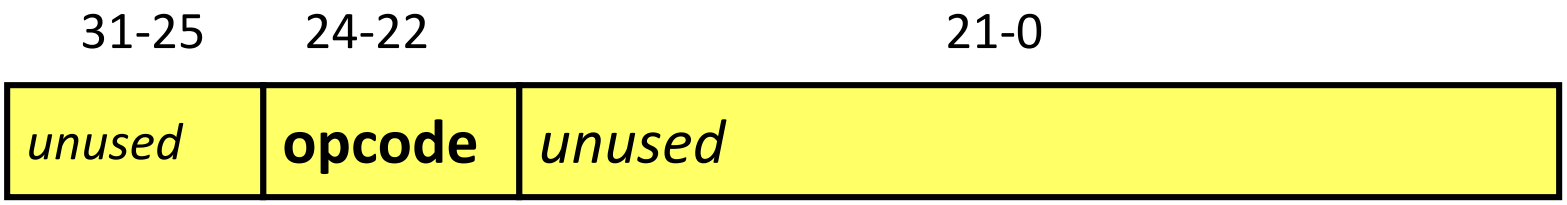
# Instruction Formats

LC2K ISA

- J-type instructions (jalr ‘101’)




- O type instructions (halt ‘110’, noop ‘111’)



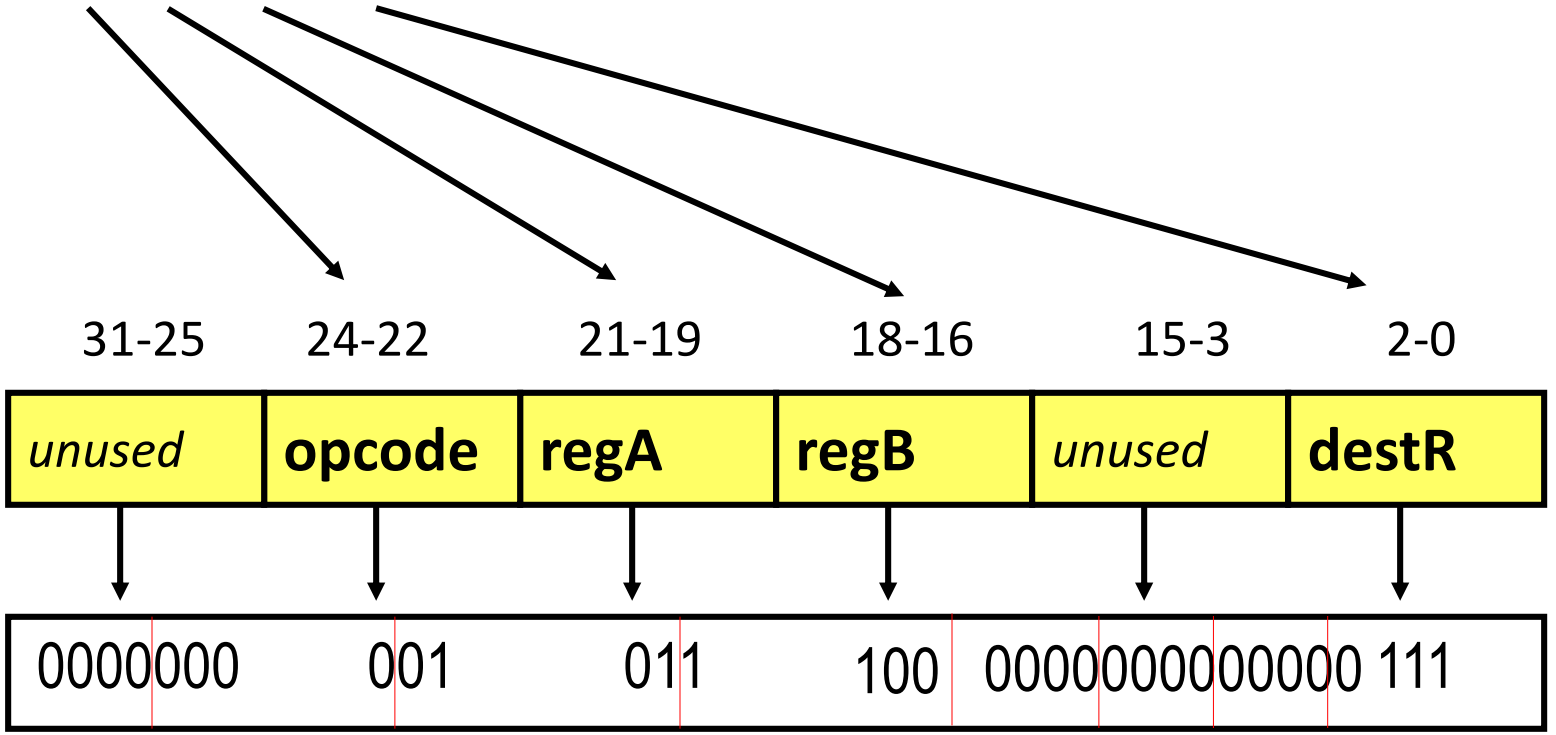
# Bit Encodings

LC2K ISA

- Opcode encodings
  - add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)
- Register values
  - Just encode the register number (r2 = 010)
- Immediate values
  - Just encode the values in 2's complement format (Remember to give all the available bits a value!!)

# Example Encoding - nor

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

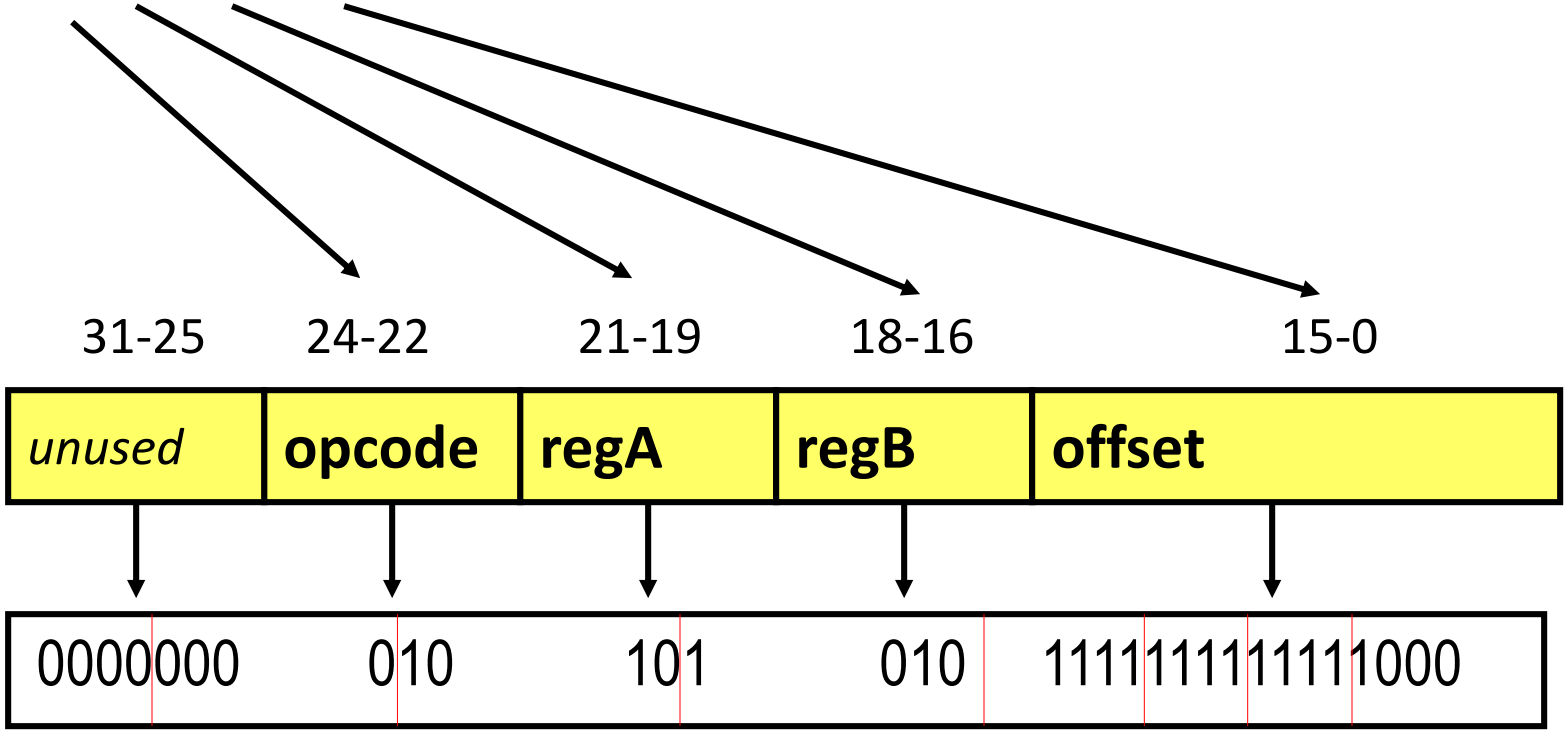



Convert to Hex → 0x005C0007

Convert to Dec → 6029319

# Example Encoding - lw

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

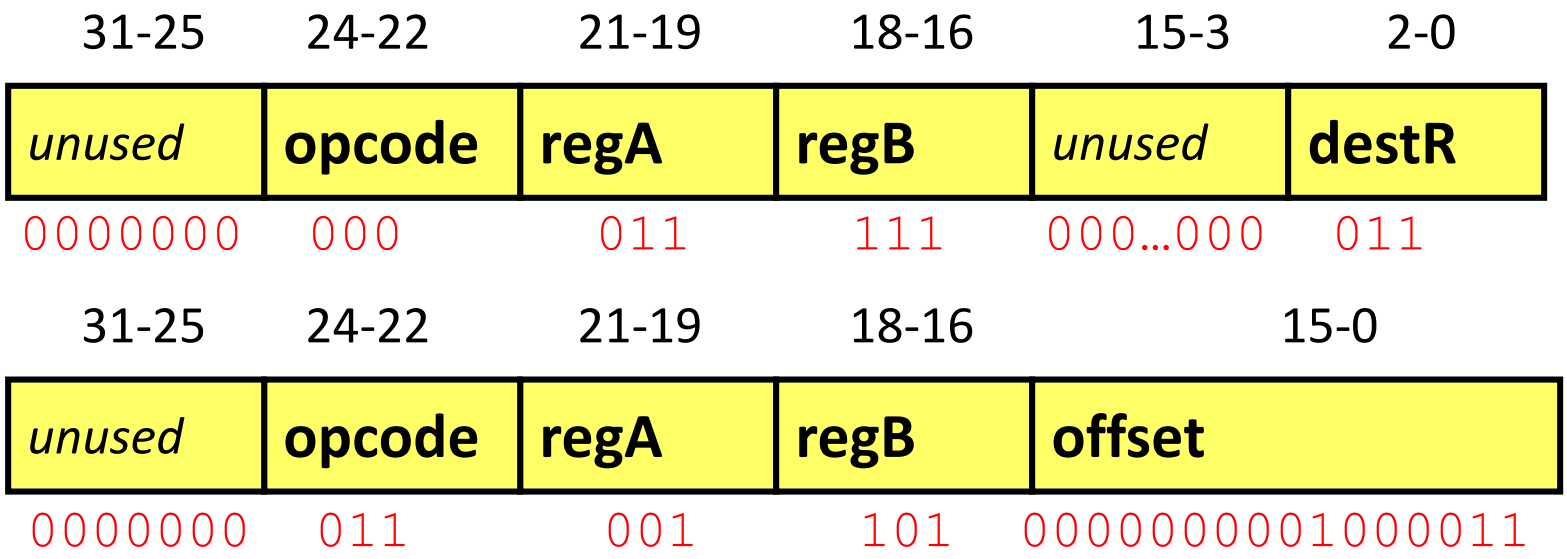



Convert to Hex → 0x00AAFFF8

Convert to Dec → 11206648

# Class Problem

- 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)



# Another way to think about the assembler

LC2K ISA

- Each line of assembly code corresponds to a number
  - "add 0 0 0" is just 0.
  - "lw  5 2 -8" is 11206648

- We only use assembly because it's easier to read.

# .fill

- 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?

# Labels in LC2K

- Labels are a way of referring to a line number in an assembly program.

```
loop   beq   3   4   end
       noop
       beq   1   3   loop
end    halt
```

- Here **loop** is 0 and **end** is 3.

- What are the values of the labels here?

```
loop   beq   3   4   end
       add   3   3   3
tom    noop
       beq   1   3   loop
end    halt
```

# Labels in LC2K



- Labels are used in lw/sw instructions or beq instruction
- For lw or sw instructions, the assembler should compute offsetField to be equal to the address of the label
  - i.e. offsetField = address of the label
- For beq instructions, the assembler should translate the label into the numeric offsetField needed to branch to that label
  - i.e. PC+1+ offsetField = address of the label

# Class problem

LC2K ISA

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

- Convert the first line into 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.

Project 1a

# Project 1a: Create an LC2K assembler

- Takes assembly code as input
- Outputs machine code (in decimal)

# Example from the spec

Project 1a

spec.as

```
1           lw      0       1       five    load reg1 with 5 (symbolic address)
2           lw      1       2       3       load reg2 with -1 (numeric address)
3   start   add     1       2       1       decrement reg1
4           beq     0       1       2       goto end of program when reg1==0
5           beq     0       0       start   go back to the beginning of the loop
6           noop
7   done    halt                            end of program
8   five    .fill   5
9   neg1    .fill   -1
10  stAddr  .fill   start                   will contain the address of start (2)
11
```

And here is the corresponding machine language:

spec.as's corresponding machine language, with addresses and hex | Your output shoul

```
1   !! Your output should only include the machine code in decimal !!
2   !! The address and hex information is for your understanding   !!
3   !! See spec.mc.correct for what your output should look like   !!
4   (address 0): 8454151 (hex 0x810007)
5   (address 1): 9043971 (hex 0x8a0003)
6   (address 2): 655361 (hex 0xa0001)
7   (address 3): 16842754 (hex 0x1010002)
8   (address 4): 16842749 (hex 0x100fffd)
9   (address 5): 29360128 (hex 0x1c00000)
10  (address 6): 25165824 (hex 0x1800000)
11  (address 7): 5 (hex 0x5)
12  (address 8): -1 (hex 0xffffffff)
13  (address 9): 2 (hex 0x2)
```

Project 1a

# Project "hints" for P1a

- There is an example output at the start of section 3.
  - Read it carefully and be sure *every single bit of output* makes perfect sense.
    - If you don't understand how to do it well enough to do by hand, you can't write a program to do it!

spec.as

```
1          lw      0       1       five    load reg1 with 5 (symbolic address)
2          lw      1       2       3       load reg2 with -1 (numeric address)
3  start   add     1       2       1       decrement reg1
4          beq     0       1       2       goto end of program when reg1==0
5          beq     0       0       start   go back to the beginning of the loop
6          noop
7  done    halt                            end of program
8  five    .fill   5
9  neg1    .fill   -1
10 stAddr  .fill   start                   will contain the address of start (2)
11
```

And here is the corresponding machine language:

spec.as's corresponding machine language, with addresses and hex | Your output shoul

```
1  !! Your output should only include the machine code in decimal !!
2  !! The address and hex information is for your understanding   !!
3  !! See spec.mc.correct for what your output should look like   !!
4  (address 0): 8454151 (hex 0x810007)
5  (address 1): 9043971 (hex 0x8a0003)
6  (address 2): 655361 (hex 0xa0001)
7  (address 3): 16842754 (hex 0x1010002)
8  (address 4): 16842749 (hex 0x100fffd)
9  (address 5): 29360128 (hex 0x1c00000)
10 (address 6): 25165824 (hex 0x1800000)
11 (address 7): 5 (hex 0x5)
12 (address 8): -1 (hex 0xffffffff)
13 (address 9): 2 (hex 0x2)
```

- Two passes
  - First one is just to get the value of each label "the symbol table"
    - Do the first pass and print the table
  - Second pass, do just adds first, then add one more instruction until you have them all.
    - Test cases with just those!

Project 1a

# Good luck!

- Try to have the spec read and have the sample fully understood by this weekend.