L3_3 LC-2K ISA

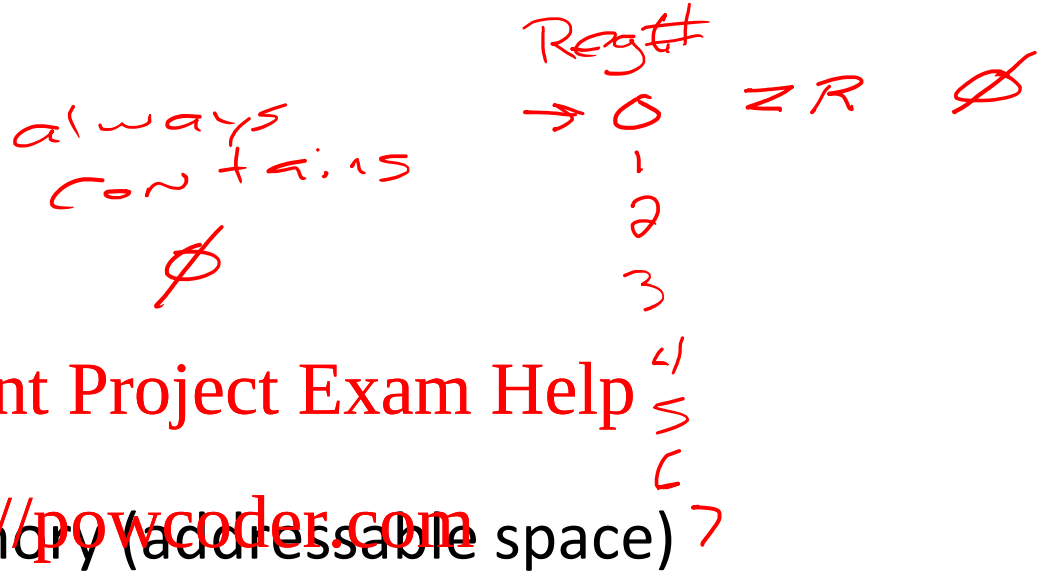
EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Recognize the set of instructions for LC-2K Architecture (ISA) and be able to describe the operations and operands for each instruction
- Ability to create simple LC-2K assembly programs, e.g., using addition and branching.
- Understand and be able to replicate the encoding (translation from assembly to machine code) of instructions for any LC-2K assembly program

LC-2K Processor

LC-2K ISA



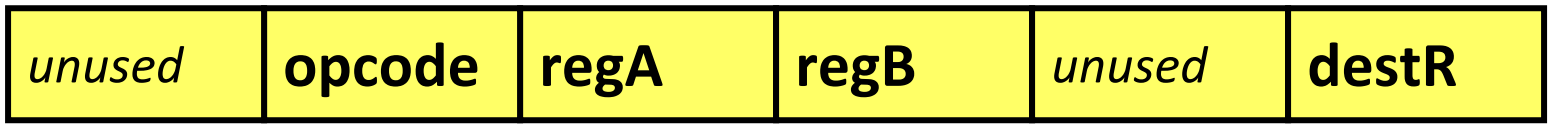
- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- supports 65536 words of memory (addressable space)
- 8 instructions in the following common categories:
  - Arithmetic: `add`
  - Logical: `nor`
  - Data transfer: `lw, sw`
  - Conditional branch: `beq`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

Instruction Encoding

LC-2K ISA

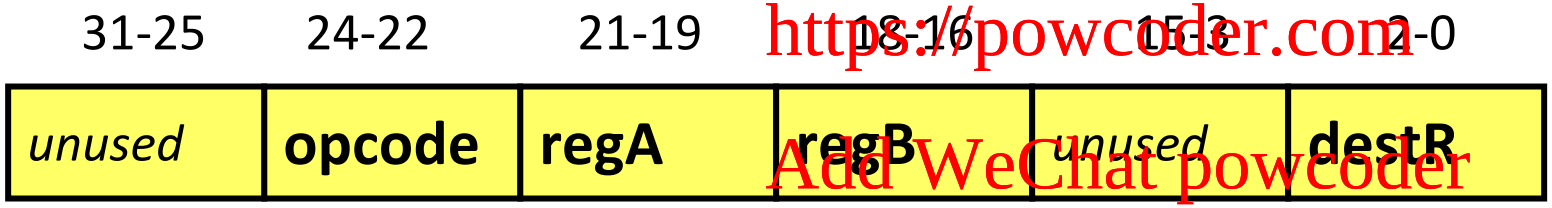
- The Instruction Set Architecture (aka Architecture) defines the mapping of assembly instructions to machine code


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

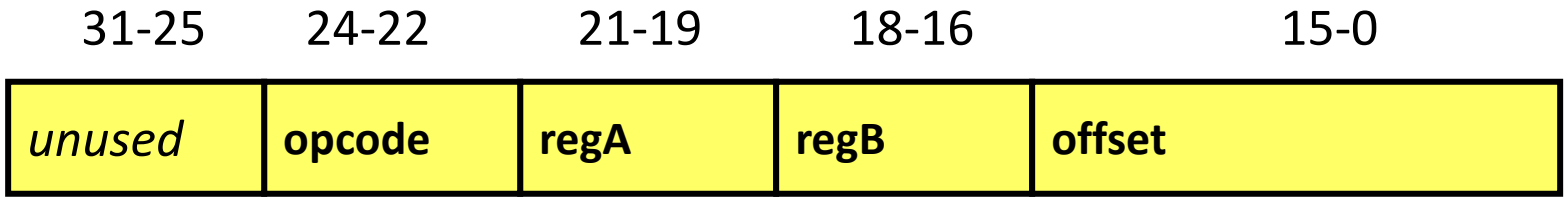



Instruction Formats – R-type, I-type

- Tells you which bit fields correspond to which part of an assembly instruction
- R-type (register) – add (opcode 000), nor (opcode 001)

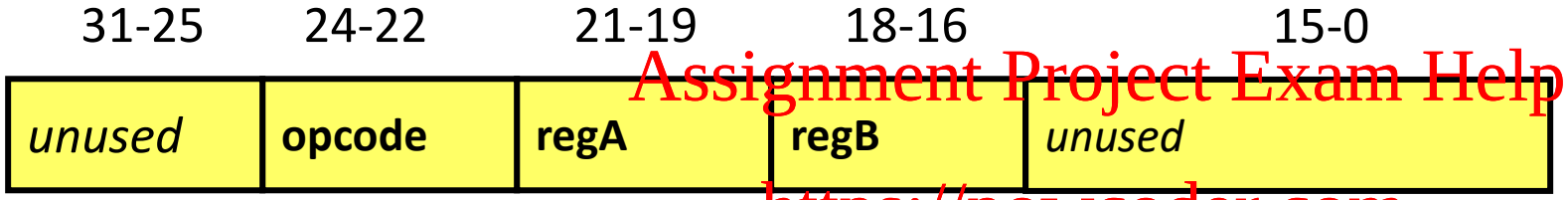


- I-type (immediate) - `lw` (opcode `010`), `sw` (opcode `011`), `beq` (opcode `100`)

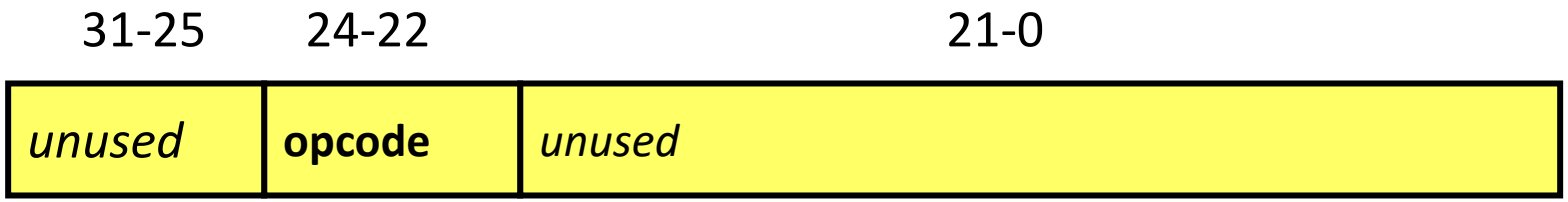


Instruction Formats – J-type, O-type

- J-type (jump) – `jalr` (opcode `101`



- O-type (???) - `halt` (opcode 110), `noop` (opcode 111)



Instruction Formats

- The Instruction Set Architecture (aka Architecture) defines the mapping of assembly instructions to machine code

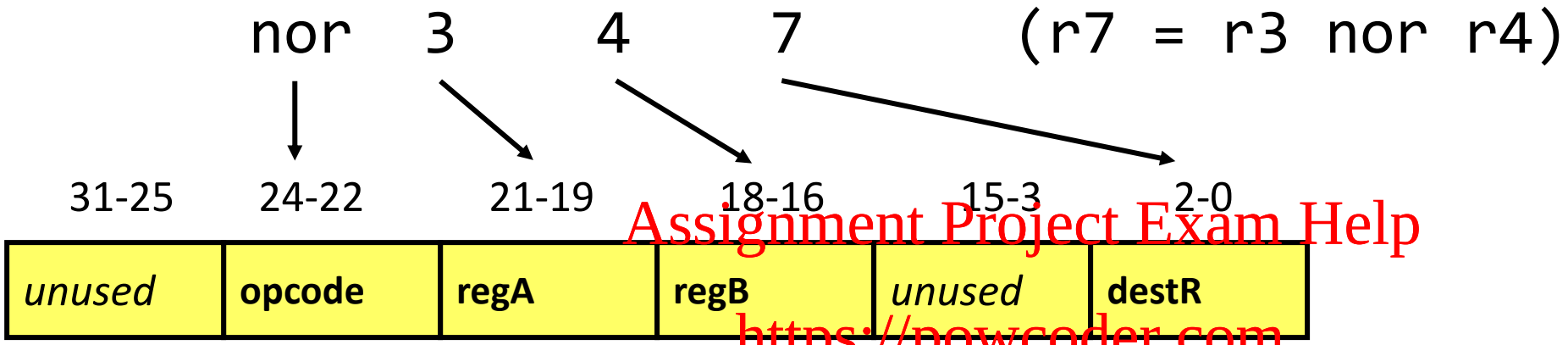
Instruction Type	Instruction	Bits 31-25	Bits 24-22	Bits 21-19	Bits 18-16	Bits 15-3	Bits 2-0
R-type	add	unused	opcode	reg A	reg B	unused	destReg
	nor						
I-type	lw					offsetField 16-bit, 2's complement number range:[-32768, 32767]	
	sw						
	beq						
J-type	jalr					unused	
O-type	halt			unused			
	noop						

Unused: all unused bits should always be 0

Bit Encodings

- Opcode binary encodings:
  - add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)
- Register operands
  - Binary encoding of register number, e.g., r2 = 2 = 010
- Immediate values
  - Binary encoding *using 2's complement values*
  - Give all available bits a value – *do not forget sign extension!*

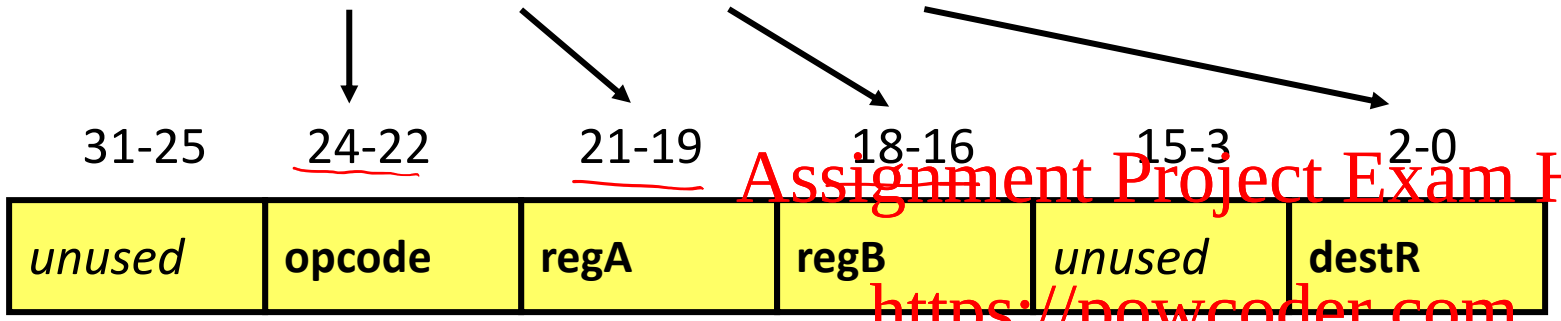
Encoding Example #1 - nor



Encoding Example #1 - nor

opcode nor = 001

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



3 = 011

4 = 100

7 = 111

0000_0000_0_01_0_11_00_0_111

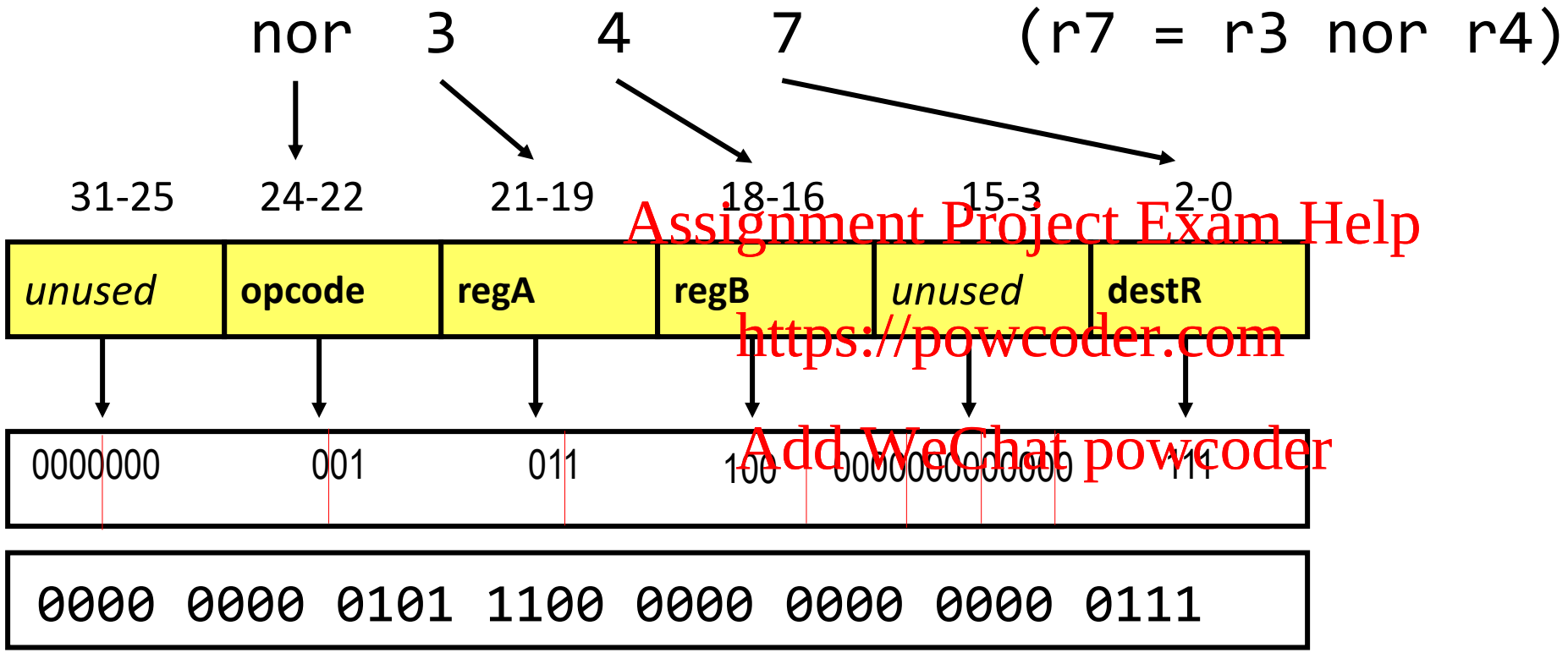
opc rA rB destR

hex 0 0 5 C 0 0 0 7

$2^{22} + 2^{20} + 2^{19} + 2^{18} + 2^2 + 2^1 + 2^0$

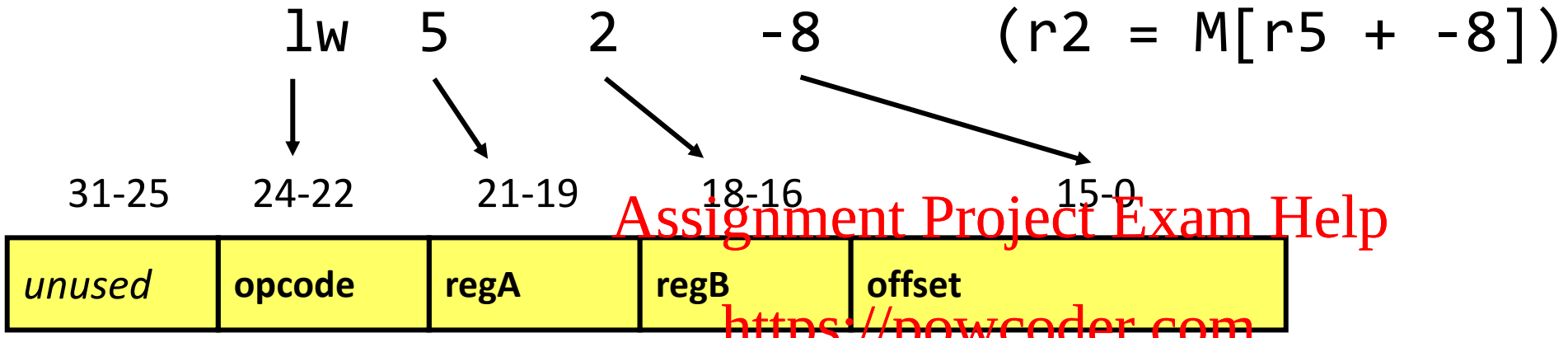
6029319

Encoding Example #1 - **nor**



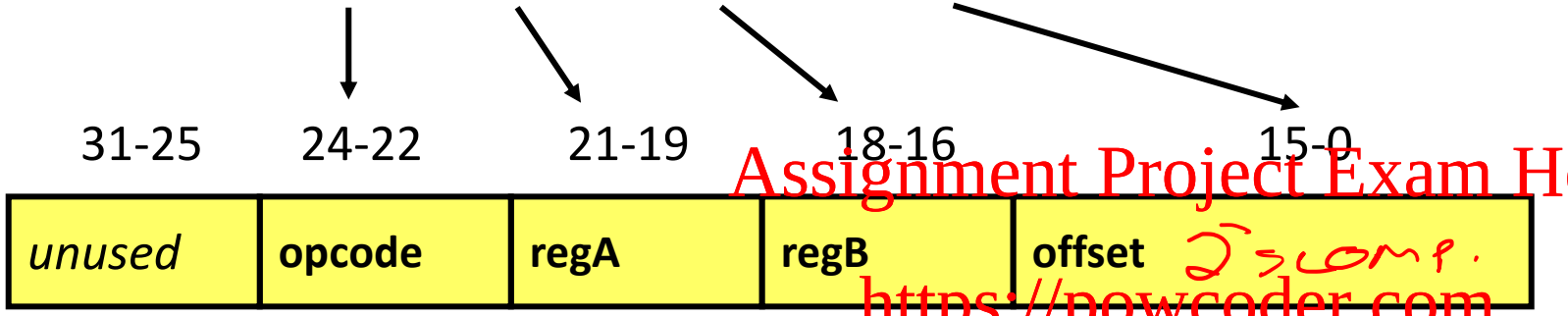
Convert to Hex → 0x005C0007
Convert to Dec → 6029319

Encoding Example #2 - `lw`



Encoding Example #2 - `lw`

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



lw = 010
5 = 101
2 = 010
-8 = ???

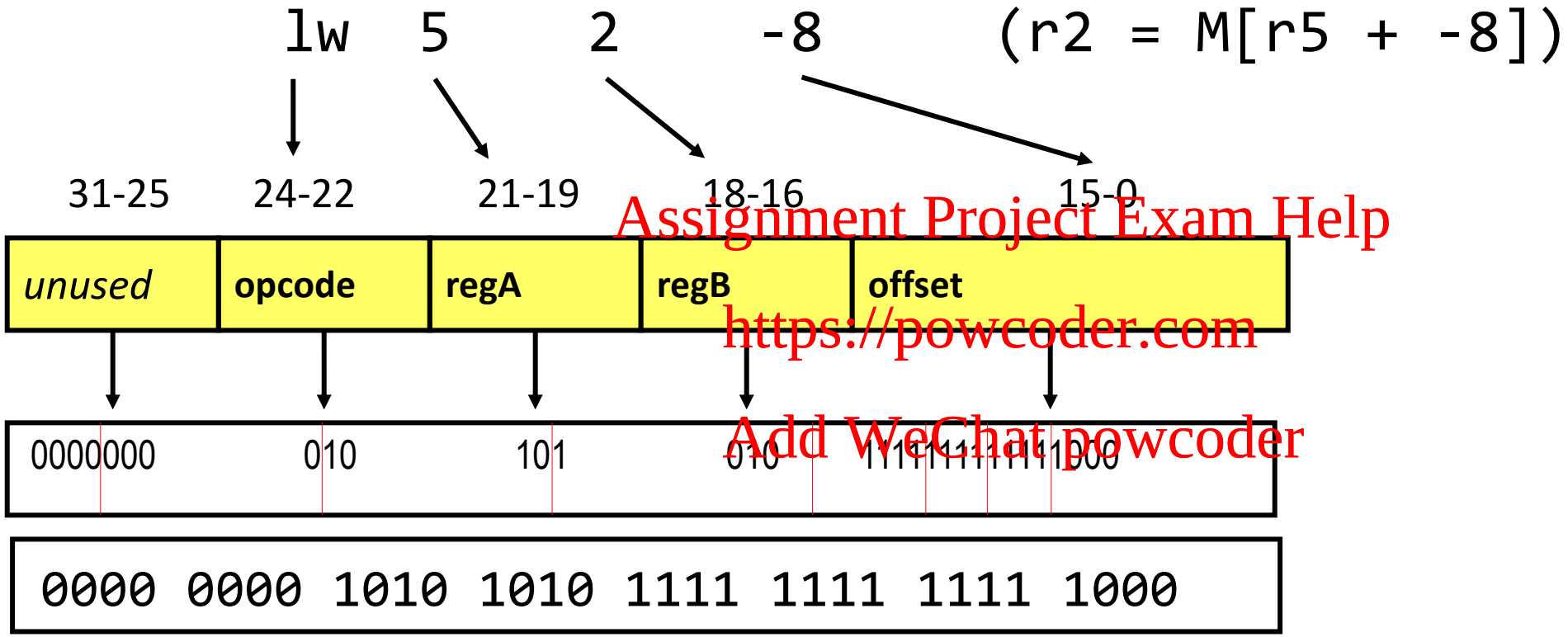
2's comp.

0000 0000 0000 1000
1111 1111 1111 0111
1
-8
1111 1111 1111 1000

0000 0010 1010 1010 1111 1111 1111 1000
oc rA rb offset

hex 0 2 A A F F F 8 →

Encoding Example #2 - `lw`



Convert to Hex → 0x00AAFFF8
Convert to Dec → 11206648

Encoding Example #3 - **add**

- Compute the encoding in Hex for:

```
add   3    7    3      (r3 = r3 + r7)      (add = 000)
```

Encoding Example #3 - **add**

- Compute the encoding in Hex for:

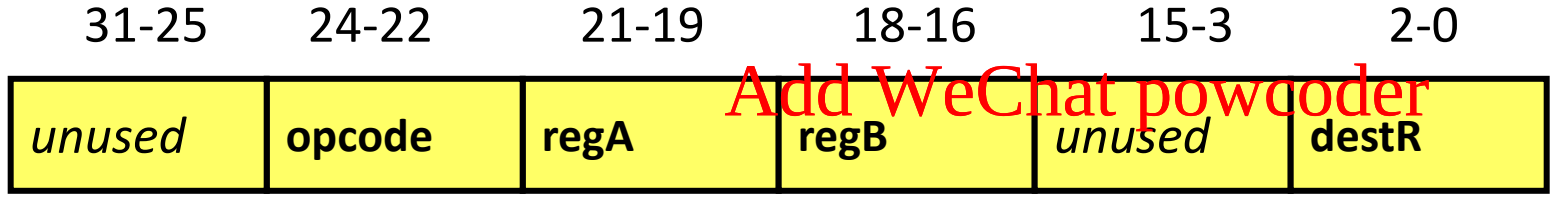
```
add   3    7    3      (r3 = r3 + r7)      (add = 000)
```

Encoding Example #3 - **add**

add 000
3 011
7 111

- Compute the encoding in Hex for:

```
add   3    7    3      (r3 = r3 + r7)      (add = 000)
```

Convert to Hex → 0x001F0003
Convert to Dec → 2031619

Encoding Example #4 - `sw`

- Compute the encoding in Hex for:

```
sw  1   5  67      (M[r1+67] = r5)   (sw = 011)
```

Encoding Example #4 - `sw`

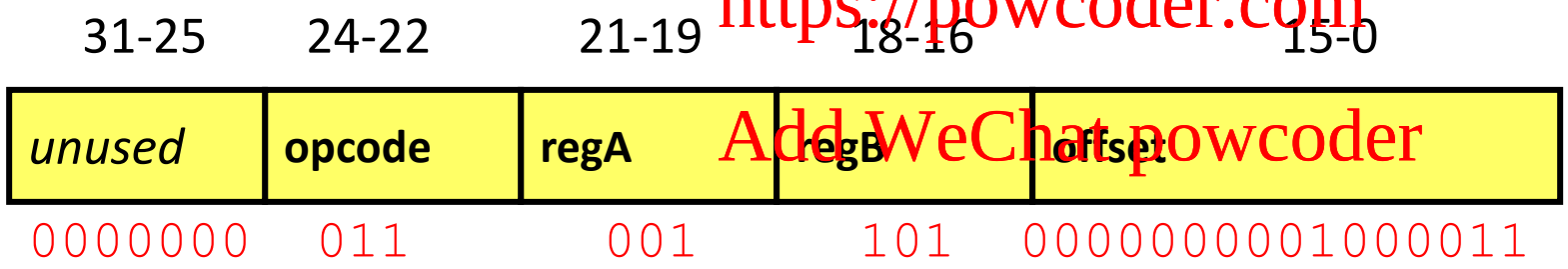
- Compute the encoding in Hex for:

```
sw  1   5  67      (M[r1+67] = r5)   (sw = 011)
```

Encoding Example #4 - `sw`

- Compute the encoding in Hex for:

```
sw   1    5   67       (M[r1+67] = r5)    (sw = 011)
```

Convert to Hex → 0x00CD0043

Convert to Dec → 13434947

Assembler, aka, P1a

- Each line of assembly code corresponds to a number
  - “add 0 0 0” is just 0.
  - “lw 5 2 -8” is 11206648
- Assembly code is how people write instructions for an ISA
  - We only use assembly because it’s easier to read.
- Assembly code must be assembled (instructions encoded) to machine code for execution

Assembler Directive - `.fill`

- You might want a number to be, well, a number.
  - Data for `lw`, `sw` instructions will be added to LC-2K assembly code file

- `.fill` tells the assembler to put a number instead of an instruction

- The syntax (to have a value of 7) is just `.fill 7`

- Question:
  - What do `.fill 7` and `add 0 0 7` have in common?

Assembler Directive - `.fill`

- You might want a number to be, well, a number.
  - Data for `lw`, `sw` instructions will be added to LC-2K assembly code file

- `.fill` tells the assembler to put a number instead of an instruction

- The syntax (to have a value of 7) is just `.fill 7`

- Question:
  - What do `.fill 7` and `add 0 0 7` have in common?

They have the same value in machine code: 7 (decimal) 111 (binary)
really `0000 0000 0000 0000 0000 0000 0000 0111`

Labels in LC-2K

- 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

Labels in LC-2K – Example #1

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

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

Labels in LC-2K – Example #1

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

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

Labels in LC-2K – Example #1

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

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


Replacing use of labels with values

```
loop  is address 0
end is address 3
```

1st pass

Addresses instructions

```
0   loop   beq   3   4   2
1          noop
2          beq   1   3   -3
3   end    halt
```

2nd pass

Program in LC-2K – Example

1. Encode program instructions 2. What does this program do?

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

Program in LC-2K – Example

1. Encode program instructions 2. What does this program do?

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

Program in LC-2K – Example

1. Encode program instructions 2. What does this program do?

test.as

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

test.mc

```
8454148
589825
12648452
25165824
1
```

Logistics

- This is the final of 3 videos for lecture 3
  - L3_1 – ISAs – Instructions and Memory
  - L2_2 – Two's Complement
  - L2_3 – LC-2K ISA
- There are two worksheets for lecture 3
  1. Addressing and 2's complement
  2. LC-2K program encoding
- Complete the participation quiz for lecture 3 on Canvas
  - Due by 9/13 at 11:59 pm