L3_3 LC-2K ISA

Assignment Project Exam Help

EECS 370 – Introduction to Computer Organization – Fall 2020

Add WeChat powcoder

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

Reg#

always contains $\emptyset$

→ 0 ZR $\emptyset$
1
2
3
4
5
6
7

- 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
```


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

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`)

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

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

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

Instruction Formats – J-type, O-type

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

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

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

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

Instruction Formats

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

<table>
<tr><th>Instruction Type</th><th>Instruction</th><th>Bits 31-25</th><th>Bits 24-22</th><th>Bits 21-19</th><th>Bits 18-16</th><th>Bits 15-3</th><th>Bits 2-0</th></tr>
<tr><td rowspan="2">R-type</td><td>add</td><td rowspan="8">unused</td><td rowspan="8">opcode</td><td rowspan="6">reg A</td><td rowspan="6">reg B</td><td rowspan="2">unused</td><td rowspan="2">destReg</td></tr>
<tr><td>nor</td></tr>
<tr><td rowspan="3">I-type</td><td>lw</td><td colspan="2" rowspan="3">offsetField
16-bit, 2’s complement number
range:[-32768, 32767]</td></tr>
<tr><td>sw</td></tr>
<tr><td>beq</td></tr>
<tr><td>J-type</td><td>jalr</td><td colspan="2">unused</td></tr>
<tr><td rowspan="2">O-type</td><td>halt</td><td colspan="4" rowspan="2">unused</td></tr>
<tr><td>noop</td></tr>
</table>

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`


```
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**

Encoding Example #1 - **nor**

opcode nor = 001

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

3 = 011
4 = 100
7 = 111

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

21

$0000\ 000\underbrace{0\ 01}_{op}\underbrace{0\ 11}_{rA}\underbrace{100}_{rB}\ 0000\ 0000\ 0000\ 0\underbrace{111}_{destR}$

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**


```
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


```
0000 0000 0101 1100 0000 0000 0000 0111
```

Convert to Hex → 0x005C0007

Convert to Dec → 6029319

Encoding Example #2 - `lw`

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

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

Encoding Example #2 - `lw`

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

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset** 2's comp.

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

0000 0000 0000 0000 0000 0000 0000 1000
1111 1111 1111 1111 1111 1111 1111 0111
1

1111 1111 1111 1111 1111 1111 1111 1000

$\boxed{-8}$

$0000\ 0000\ \underbrace{010}_{oc}\ \underbrace{101}_{rA}\ \underbrace{010}_{rb}\ \underbrace{1111\ 1111\ 1111\ 1000}_{offset}$

hex 0 0 A A F F F 8 →

Encoding Example #2 - `lw`

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

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

`0000 0000 1010 1010 1111 1111 1111 1000`

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)
```


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

00 0000000 000 0 1 1 1 1 1 0000 0000 0000 0011

dreg

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`

sw = 3 011

1 = 001

5 = 101

- Compute the encoding in Hex for:

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

$\boxed{67}$

1 0

64 + 2 + 1

0000000 011 001 101 0000000001000011

Encoding Example #4 - `sw`

- Compute the encoding in Hex for:

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


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

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


loop : 0
end : 3

beq : 100
3 011
4 100

100 | 001 | 011 | ____
op | rA | rB | off

beq 3 4 ~~3~~ off + PC+1 = 3
off + 0 + 1 = 3
off + 1 = 3
off = 2

PC + 1 + off = 0
2 + 1 + off = 0 off = -3

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?

```
0  loop   lw    0  1  one   4
1         add   1  1  1
2         sw    0  1  one   4
3         halt
4  one    .fill 1
```

1st Pass

loop : 0

one : 4

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