CS 264 Systems **Spring 2020**
http://neilklingensmith.com/teaching/loyola/cs264-s2020

Homework 4

Due: April 27, 2020

Name:

In this homework, you will be extending the Littlecpu to add some extra instructions. The instruction encodings that I have defined are at the end of this document. You can use them for your reference.

1. (25 points) The `ADDI` instruction: adding an immediate value to a register.
   (a) (5 points) Create a new instruction called `ADDI` which adds a hard-coded immediate value to register `RD`. Draw the instruction encoding in the space below. Examples of how `ADDI` could be used in a program:

```
ld r0,[r1+r2] ; Read a value out of memory
addi r0,#5    ; r0 <- r0 + 5
st r0,[r1+r2] ; Store incremented value back to memory
```

   (b) (10 points) Assemble the program above by hand. Write out the binary and hex representation of each instruction.

Instruction Mnemonic	**Binary Encoding**	**Hex Encoding**
`ld r0,[r1+r2]`		
`addi r0,#5`		
`st r0,[r1+r2]`		

   (c) (5 points) Explain what changes need to be made in the control path of the Littlecpu in order to support the `ADDI` instruction.

(d) (5 points) Can your `ADDI` instruction be easily extended to other ALU operations like `SUBI`, `ANDI`, etc? What modifications would you need to make to support a broader range of immediate ALU instructions?

2. (25 points) The `CMP` instruction

(a) (5 points) Create a new instruction called `CMP` that compares two operands `RS` and `RT`. It should work by subtracting two operands without writing the result back to the register file. Example of how the `CMP` instruction could be used in a program:

```
ldi r0,#100          ; Get immediate value to compare to in r0
ldi r1,#85           ; Get address of a variable in r1
ldi r2,#0            ; Zero out r2
ld r1, [r1+r2]       ; Load value of variable into r1
cmp r0,r1  ; r0 - r1 ; Compare the variable value to 100
```

(b) (15 points) Assemble the program above by hand. Write out the binary and hex representation of each instruction.

Instruction Mnemonic	**Binary Encoding**	**Hex Encoding**
`ldi r0, #100`		
`ldi r1, #85`		
`ldi r2, #0`		
`ld r1, [r1+r2]`		
`cmp r0, r1`		

(c) (5 points) Explain what changes need to be made in the control path of the Littlecpu in order to support the `CMP` instruction.

3. (25 points) The `JMP` (jump to immediate) instruction

   (a) (5 points) Create a new instruction called `JMP` where the jump target address is hardcoded into the instruction as an immediate value. Draw the encoding of the `JMP` instruction below. Example of how the `JMP` instruction could be used in a program:

```
0x10: ldi r0,#10
0x11: ldi r1,#0
0x12: ldi r3, #200
0x13: st r1,[r3+r0]
0x14: subi r0,#1
0x15: jmp 0x13      ; jmp back to beginning of loop
```

(b) (15 points) Assemble the program above by hand. Write out the binary and hex representation of each instruction.

Instruction Mnemonic	**Binary Encoding**	**Hex Encoding**
`ldi r0, #10`		
`ldi r1, #0`		
`ldi r3, #200`		
`st r1, [r3+r0]`		
`subi r0, #1`		
`jmp 0x13`		

(c) (5 points) Explain what changes need to be made in the control path of the Littlecpu in order to support the `JMP` instruction.

4. (15 points) Implement the `JR` instruction in the Littlecpu datapath.

5. (15 points) Implement the `ADDI` instruction in the Littlecpu datapath.

Instruction Encodings for the Littlecpu

ALU Instructions ALU instructions perform ADD, SUB, AND, etc. operations. The inputs are taken from the register file. Example instructions are:

```
add r0,r3,r2 ; r0 <- r3 + r2
sub r0,r0,r1 ; r0 <- r0 - r1
and r2,r2,r1 ; r0 <- r0 & r1 (bitwise AND)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	RD		RS		RT		0	ALUOP		

The ALU operations are defined in the table below. These are hard-coded by the ALU block in CircuitVerse. The ALU block in CircuitVerse tells us what operation it is performing when you change the CTRL input.

ALUOP	**Operation**
000	AND
001	OR
010	ADD
011	Not Defined
100	NAND
101	NOR
110	SUB
111	COMPARE

LD Instruction Loads a byte out of the data memory from the effective address `RS+RT`.

```
ld r3,[r0,r1] ; r3 <- mem(r0+r1)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	RD		RS		RT		0	0	1	0

ST Instruction Stores a byte to the data memory at the effective address `RS+RT`.

```
ld r3,[r0,r1] ; r3 <- mem(r0+r1)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	RD		RS		RT		1	0	1	0

LI Instruction Loads a hard-coded immediate value into register `RD`.

```
li r0, #0x48   ; r0 <- 0x48
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	RD		imm8							

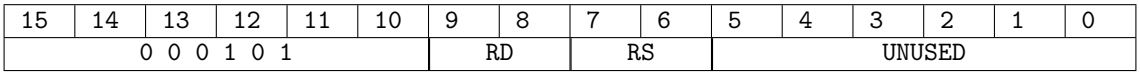
JR Instruction Jump to `RD` in the program.

```
jr r0 ; pc <- r0
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	RD		CONDITION							

JLR Instruction Call function pointed to by `RS`. Save return address in `RD`.

```
jlr r0,r3 ; r3 <- return address, pc <- r0
```



NOP Instruction Do nothing

```
nop
```