Project 2: The mcu91 ISA

This is not all you need to do the project, unless you're already an experienced hardware designer. **Open up the project details too… that will guide you through it.**

Instruction formats

mcu91 instructions are **20 bits long.** There are three instruction formats: R, I, and J.

- The top bit is 0 for R-type and I-type instructions, and 1 for J-type.
  - The opcode field is **a different size** in the two formats!
  - The "opcode" column in the tables below is `0xxxxx` for R/I-type instructions, and `1xxx` for J-type instructions.
- All three formats have an immediate, but **the immediate is different sizes.**
  - To distinguish them, they're named according to their size (`imm5` , `imm8` , and `imm16`).
- The `rd` , `rs` , and `rt` fields hold register numbers.
  - So if `rd == 4` , then that means `r4` .
  - In R-type instructions, the result is written to register `rd` .
  - In I-type instructions, the result is written to register `rt` .

R-Type Instruction

19	18 14	13 11	10 8	7 5	4 0
0	opcode	rs	rt	rd	imm5

I-Type Instruction

19	18 14	13 11	10 8	7 0
0	opcode	rs	rt	imm8

J-Type Instruction

19	18 16	15 0
1	opcode	imm16

The mcu91 instruction set

- They are listed more or less in the order I recommend you try implementing them.
- The Mnemonic column shows how the instruction is written.
- The Operation column shows how the instruction works. **That's what you implement!**
- The `<-` means "store the value on the right into the thing on the left."
- `sxt()` and `zxt()` mean sign-extend and zero-extend. (There are components to do this in Logisim.)
- `x[4:0]` or whatever means "bits 4, 3, 2, 1, and 0 of x".
- `REG[x]` means "the register file whose number is given by x."
- `RAM[x]` means "the word in RAM at memory address x."

Basic instructions

`lui` means "load upper immediate." Just like in MIPS, this loads the immediate into the upper half of the register, and sets the lower half to 0. This is used in combination with `ori` to load a 16-bit immediate.

Format	Opcode	Mnemonic	Operation
R	`000000`	`halt`	stop CPU and turn on HALT LED
R	`000001`	`put rs`	`display <- REG[rs]`
I	`000010`	`li rt, imm`	`REG[rt] <- zxt(imm8)`
I	`000011`	`lui rt, imm`	`REG[rt] <- imm8 << 8`

Memory access

Format	Opcode	Mnemonic	Operation
R	`000100`	`ld rd, [rs+imm5]`	`REG[rd] <- RAM[REG[rs] + zxt(imm5)]`
R	`000101`	`st [rs+imm5], rt`	`RAM[REG[rs] + zxt(imm5)] <- REG[rt]`

When implementing these, think to yourself: do you really need another component to do the addition? ;)

ALU operations

Both `shr` and `sri` should use **logical** right shifts.

Be careful about sign- vs. zero-extension on the I-type instructions!

Format	Opcode	Mnemonic	Operation

Format	Opcode	Mnemonic	Operation
R	`000110`	`add rd, rs, rt`	`REG[rd] <- REG[rs] + REG[rt]`
R	`000111`	`sub rd, rs, rt`	`REG[rd] <- REG[rs] - REG[rt]`
R	`001000`	`and rd, rs, rt`	`REG[rd] <- REG[rs] & REG[rt]`
R	`001001`	`or rd, rs, rt`	`REG[rd] <- REG[rs] | REG[rt]`
R	`001010`	`not rd, rs`	`REG[rd] <- ~REG[rs]`
R	`001011`	`shl rd, rs, rt`	`REG[rd] <- REG[rs] << REG[rt][3:0]`
R	`001100`	`shr rd, rs, rt`	`REG[rd] <- REG[rs] >> REG[rt][3:0]`
I	`001101`	`adi rt, rs, imm8`	`REG[rt] <- REG[rs] + sxt(imm8)`
I	`001110`	`sbi rt, rs, imm8`	`REG[rt] <- REG[rs] - sxt(imm8)`
I	`001111`	`ani rt, rs, imm8`	`REG[rt] <- REG[rs] & zxt(imm8)`
I	`010000`	`ori rt, rs, imm8`	`REG[rt] <- REG[rs] | zxt(imm8)`
I	`010001`	`sli rt, rs, imm8`	`REG[rt] <- REG[rs] << imm8[3:0]`
I	`010010`	`sri rt, rs, imm8`	`REG[rt] <- REG[rs] >> imm8[3:0]`

Control flow

Function return addresses are not stored in registers *or* in regular data memory. There's a separate component for them! That's `STK[]` and `sp` below.

Format	Opcode	Mnemonic	Operation
I	`010011`	`blt rs, rt, imm8`	`if(REG[rs] < REG[rt]) { PC <- PC + sxt(imm8) }`
I	`010100`	`bge rs, rt, imm8`	`if(REG[rs] >= REG[rt]) { PC <- PC + sxt(imm8) }`
I	`010101`	`beq rs, rt, imm8`	`if(REG[rs] == REG[rt]) { PC <- PC + sxt(imm8) }`
I	`010110`	`bne rs, rt, imm8`	`if(REG[rs] != REG[rt]) { PC <- PC + sxt(imm8) }`
I	`010111`	`bez rs, imm8`	`if(REG[rs] == 0) { PC <- PC + sxt(imm8) }`

Format	Opcode	Mnemonic	Operation
I	`011000`	`bnz rs, imm8`	`if(REG[rs] != 0) { PC <- PC + sxt(imm8) }`
R	`011001`	`jr rs`	`PC <- REG[rs]`
J	`1000`	`j imm16`	`PC <- imm16`
J	`1001`	`call imm16`	`PC <- imm16` `STK[sp] <- PC + 1` `sp <- sp + 1`
R	`011010`	`ret`	`PC <- STK[sp - 1]` `sp <- sp - 1`