CSE 141L Milestone 1

John Adams A16499049

# Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

* Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
* Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

John P Adams

# 0. Team

John Adams.

# Introduction

Name: TMR (too many registers)

Philosophy: Use specialized registers so you can do more “stuff” without having to specify where it comes from.

Goals: make a cpu that was:

* easy to code for (lots of registers, designed with for loops in mind, 8-bit literals, many math operations)
* entirely from scratch, because why would you learn how to use an api when you can spend twice as long debugging your own code.

My cpu is a Load-Store (register-register) architecture. Although by using one of the address registers, it would be possible to implement a stack in software with only 2 instructions for push (stor b {reg}, incr b) and pop (load b {reg}, decr b).

# Architectural Overview

A diagram of a computer

Description automatically generated

Yes, I got it to build in Quartus.

# Machine Specification

## Instruction formats (my formats lol)

|  |  |  |
| --- | --- | --- |
| **TYPE** | **FORMAT** | **CORRESPONDING INSTRUCTIONS** |
| A | 5-bit OP code, 4-bit val | litl, lith, seth |
| B | 5-bit OP code, 4-bit reg | movc, movd,  movm, movn, movx, movy,  mova, movb, movi, movj,  movk, movl, movz, movp,  bizr, bnzr, incr, decr, flip |
| C | 5-bit OP code, 4-bit operand | mthr, mths, func |
| A’ | 5-bit OP code, 1-bit reg, 3-bit val | jizr, jnzr, lslc, lsrc |
| B’ | 5-bit OP code, 1-bit reg, 3-bit reg | load, stor |

## Operations

Preface: I don’t have any bit-breakdown to do (there is only one way to .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **TYPE** | **BIT BREAKDOWN** | **EXAMPLE** | **NOTES** |
| amp  logical and | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0000) | # let x hold b11111100  # let y hold b00111111  mth r amp  # r now holds b00111100 | For all operations, source registers are x and y, and the result is stored in either r or s register |
| lor  logical or | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0001) | # let x hold b11111100  # let y hold b00111111  mth r lor  # r now holds b11111111 |  |
| flp  logical not | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0010) | # let x hold b11111100  mth r flp  # r now holds b00000011 |  |
| eor  logical exclusive or | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0011) | # let x hold b11111100  # let y hold b00111111  mth r eor  # r now holds b11000011 |  |
| rsc  right shift carry | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0100) | # let x hold b10111100  # let y hold b00111111  mth r rsc  # r now holds b11011110 | res = {y[0],x[7:1]} |
| lsc  left shift carry | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0101) | # let y hold b00111111  mth r lsc  # r now holds b00011111 | res = {x[6:0,]y[7]} |
| rol  rotate left | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0110) | # let x hold b11101000  # let y hold b00000011  mth r rol  # r now holds b01000111 | rotate x by value in y[2:0] |
| add  algebraic add | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (0111) | # let x hold b00001111  # let y hold b00000001  mth r add  # r now holds b00010000 |  |
| sub  algebraic subtract | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1000) | # let x hold b00010110  # let y hold b00000111  mth r sub  # r now holds b00001111 |  |
| eql8  check for byte equality | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1001) | # let x hold b10101000  # let y hold b10101111  mth r eql8  # r now holds b00000000 |  |
| eql5  check for upper 5-bit equality | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1010) | # let x hold b10101000  # let m hold b10101111  mth r eql5  # r now holds b00000001 | tests for x[7:4] = m[7:4]  **note** this is the only math instruction that uses an register outside of x,y |
| revx  reverse byte x | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1011) | # let x hold b11110000  mth r revx  # r now holds b00001111 |  |
| revy  reverse byte y | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1100) | # let y hold b10100011  mth r revy  # r now holds b11000101 |  |
| parx  compute x parity | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1101) | # let x hold b00000111  mth r parx  # r now holds b00000001 |  |
| pary  compute y parity | C | 4-bit OP code (1101) 1-bit result reg (X) 4-bit operation (1110) | # let y hold b10101010  mth r pary  # r now holds b00000000 |  |
| seth  set high | A |  |  |  |
| lslc  logical shift left with carry | A’ |  |  |  |
| lsrc  logical shift right with carry | A’ |  |  |  |
| flip  bitwise xor | B |  |  |  |

## Internal Operands

There are 16 registers (since I needed 11, I added the other 5 to use up the rest of the 4-bits needed to pick between more than 8)

Several registers are special-purpose.

address: a, b - specify the memory address for load and store instructions

math: x, y - primary inputs for ALU

result: r, s - read-only result registers from ALU

bitwise: m, n - registers for bit-wise operations, and additional ALU input parameters (for special operations)

literal: l - location for literal value instructions

branch: z - branch target

pc: p - program counter

generic: c, d, i, j, k - generic registers for counters and other things.

## Control Flow (branches)

Note: cycles are a measure of jumping to a **user-specified address** (i.e. directly from a literal value).

There are two conditional branches (branch if zero and branch if not zero) which can update the lower 8-bits of the program counter to the value in the branch (z) register based on the value of any register (range:256, precision:1, cycles: 4).

There are 4 ways to do a jump

* jmp instructions add or subtract (4’b \* 2) from the program counter (range: 32, precision: 2, cycles: 1)
* mov instructions can copy the value of any register into the pc register’s lower 8-bits (range:256, precision:1, cycles:3)
* bank-change functions modify only the upper 2-bits of the pc register (range:1024, precision:256, cycles:1)
* long-jump functions modify the upper 2-bits of pc and load the branch (z) register into lower 8-bits (range:1024, precision:1, cycles:4)

## Addressing Modes

Memory is handled indirectly. Memory addresses have to be stored in either of the two 8-bit address registers (a, b).

Load instructions can read either address register, and store into only one of the 3-bit accessible registers (c, d, m, n, x, y, c, d) excluding the read-only result registers (r, s).

Store instructions can also read either address register, and store from only one of the 3-bit accessible registers (r, s, c, d, m, n, x, y).

# Programmer's Model [Lite]

TODO. 4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.

TODO. 4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

# Program Implementation

An example Pseudocode and Assembly Code has been filled out for you. When you submit, please delete the example along with this paragraph.

## Example Pseudocode

# function that performs division

mul\_inverse(operand):

divisor = operand

dividend = 1

result = 0

counter = 0

while counter != 16:

if dividend > divisor:

dividend -= divisor

result = (result << 1) || 1

else:

result = (result << 1)

dividend <<= 1

counter += 1

return result

## Example Assembly Code

# Do not try to understand this code. It is bogus code, but a good example of what to submit.

# loading divisor

load R0, %0010 # 0010 = location of the divisor in memory

load R1, %0100 # 0100 = location of the dividend in memory

add R0, R1, R2 # R0 + R1 => R2 adding the divisor and the dividend together

...

# more assembly code

...

# note that this may be several pages long. The teaching staff will not be verifying correctness of your assembly code for Milestone 1.

## Program 1 Pseudocode

TODO

## Program 1 Assembly Code

TODO

## Program 2 Pseudocode

TODO

## Program 2 Assembly Code

TODO

## Program 3 Pseudocode

TODO

## Program 3 Assembly Code

TODO