CSE 141L Milestone 2

Christopher Edwards, A16855415

Keisuke Hirano, A16974148

Oscar Zheng, A16880333

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.

Oscar Zheng
Keisuke Hirano
Christopher Edwards

0. Team

Oscar Zheng, Keisuke Hirano, Christopher Edwards

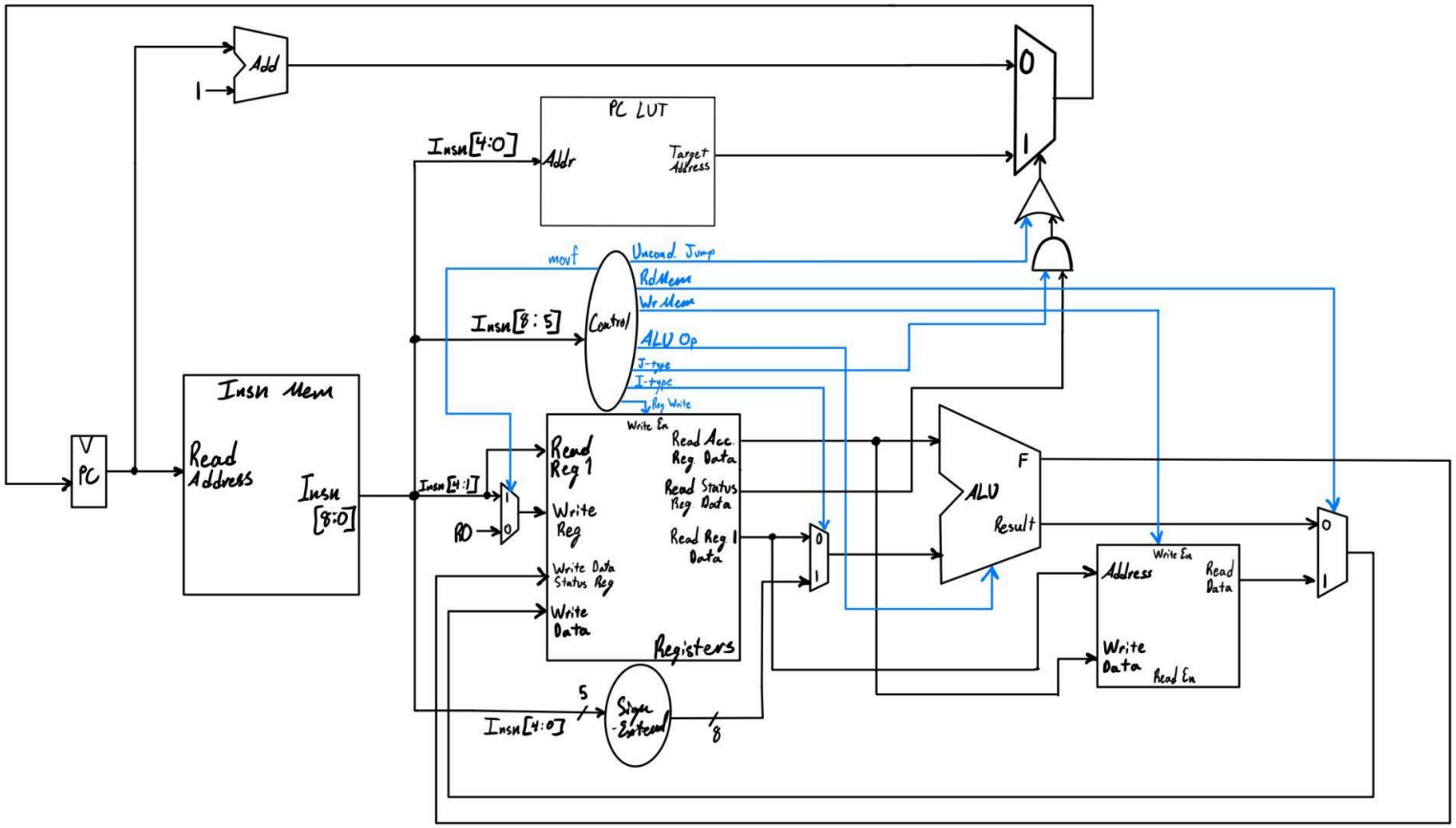
1. Introduction

TODO. Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.

Our general philosophy was creating an architecture that was capable of implementing algorithms with simple instructions that deal with computation primarily at register level without storing much in main memory. In our design, we specifically targeted programs 1 and 2 and neglected whether the ISA has the capability to solve program 3. We designed our ISA to be very RISC-oriented, in that it offers limited arithmetic instructions and not much flexibility with immediate values. It relies on significant register manipulation and moving values around. Our machine follows an accumulator design, where one register is the dedicated destination register and the first register operand. We also have a status register to hold the single flag that the ALU sets. We included two more, unusual instructions, movf and movt, in order to make data maneuverability easier in the accumulator register.

2. Architectural Overview

TODO. This must be in picture form. What are the major building blocks you expect your processor to be made up of? You must have data memory in your architecture. (Example of MIPS: https://www.researchgate.net/figure/The-MIPS-architecture_fig1_251924531)



3. Machine Specification

Instruction formats

Two example rows have been filled for you. When you submit, do not include the example types. Add rows as necessary. In your submission, please delete this paragraph.

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	4 bit opcode, 4 bit reg	add, xor, sub, ld, str, movf, movt, cmp
J	4 bit opcode, 5-bit jump offset	jmp, jc, jnn, jlt, jeq
I	4 bit opcode, 5 bit immediate	movi, lsl

Operations

An example row has been filled for you. When you submit, do not include the example type. In the name column, be sure to also add the definition of what the example actually does. For example, "lsl = logical shift left" would be an appropriate value to put in the name column. In the bit breakdown column, add in parenthesis what specific values the bits should be in order. X indicates that it will be specified by the programmer's instruction itself (i.e. specifying registers). In the example column, give an example of an “assembly language” instruction in your machine, then translate it into machine code. Add rows as necessary. In your submission, please delete this paragraph.

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
jmp = jump unconditionally	J	4 bit opcode (0000), 5 bit index of look-up table	Before: PC_LUT[3] = 0x05 PC = 0xff `jmp #3` After: PC = 0x05	jump to the absolute address given by the value in the PC LUT at the index that’s provided in the instruction
jc = jump if carry out	J	4 bit opcode (0001), 5 bit index of look-up table	Before: PC_LUT[3] = 0x05 C = 1 PC = 0xff `jc #3` After: PC = 0x05	jump if the C flag (Carry out) is set
jlt = jump less than	J	4 bit opcode (0010), 5 bit index of look-up table	Before: PC_LUT[3] = 0x05 N = 1 PC = 0xff `jlt #3`	jump if the N flag (Negative) is set

			After: PC = 0x05	
jeq = jump if equal to	J	4 bit opcode (0011), 5 bit index of look-up table	Before: PC_LUT[3] = 0x05 Z = 1 PC = 0xff `jeq #3` After: PC = 0x05	jump if the Z flag (zero) is set
jnn = jump not negative number	J	4 bit opcode (0100), 5 bit index of look-up table	Before: PC_LUT[3] = 0x05 N = 0 PC = 0xff `jnn #3` After: PC = 0x05	jump if the N flag is reset
add	R	4 bit opcode (0101), 4 bit register (XXXX)	Before: R0 = 0b0000_0001 R1 = 0b1000_0000 `add R1` After: R0 = 0b1000_0001	adds the accumulator register and given register and stores in accumulator
xor	R	4 bit opcode (0110), 4 bit register (XXXX)	Before: R0 = 0b0000_0001 R1 = 0b1000_0000 `xor R1`	xors the accumulator register and given register and stores in accumulator register

			After: R0 = 0b1000_0001	
sub	R	4 bit opcode (0111), 4 bit register (XXXX)	Before: R0 = 0b0001_0000 R1 = 0b0001_0001 `sub R1` After: R0 = 0b1111_1111 N = 1	subtracts the given register from the accumulator and stores it in the accumulator. Sets the N (negative) flag.
ld	R	4 bit opcode (1000), 4 bit register (XXXX)	Before: Memory[0x05] = 0x10 R1 = 0x05 `ldr R1` After: R0 = 0x10	load from the address given in the operand register from memory and store value in the accumulator register
str	R	4 bit code (1001), 4 bit register (XXXX)	Before: R0 = 0x10 R1 = 0x05 `str R1` After: Memory[0x05] = 0x10	store the value in the accumulator register to the address in memory given by the operand register
movf = move from reg	R	4 bit opcode (1010), 4 bit register (XXXX)	Before: R0 = 0x05 `movf R1` After: R1 = 0x05	moves information from accumulator register to the operand register
movt = move to reg	R	4 bit opcode (1011), 4 bit register (XXXX)	Before: R0 = 0x00 R1 = 0x05	moves the value in the operand register to the accumulator register

			`movt R1` After: R0 = 0x05	
cmp	R	4 bit opcode (1100), 4 bit register (XXXX)	Before: R0 = 0x05 R1 = 0x05 `cmp R1` After: Z = 1	subtracts the operand register from the accumulator register, sets the N (negative) flag if the result is 0
lsl = left shift logical	I	4 bit opcode (1101), 5 bit immediate (XXXXX)	Before: R0 = 0b0100_0001 C = 0 `lsl #2` After: R0 = 0b0000_0100 C = 1	logical left shift the value in the accumulator register by a number of iterations given by the immediate value operand. Inputs the last shifted-out bit into the C flag
rsl = right shift logical		4 bit opcode (1111), 5 bit immediate (XXXXX)	Before: R0 = 0b0100_0001 C = 0 `rsl #2` After: R0 = 0b0001_0000	logical right shift the value in the accumulator register by a number of iterations given by the immediate value operand.
movi = move immediate	I	4 bit opcode (1110), 5 bit immediate (XXXXX)	Before: R0 = 0x00 `movi #13` After: R0 = 0x0D	moves immediate to the accumulator register

Internal Operands

TODO. How many registers are supported? Is there anything special about any of the registers (e.g. constant, accumulator), or all of them general purpose?
The number of registers we support are 16 registers. Even though there are 5 bits, because of the limitations that were given, we think that this is an efficient way of using the registers. These registers would be accumulator registers.

Control Flow (branches)

TODO. What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? How do you accommodate large jumps?
We support 5 types of branches which are: jump unconditionally, jump when equal, jump when less than, jump when carry out, and jump when non negative. There is only one flag which is set in different scenarios; the flag being set signifies different things dependent on the instruction before it. The cmp instruction would set the flag to true if both registers were equal, sub would set to true if the operator register is greater than the accumulator, and lsl would set the flag to the bit that was shifted out.
The addresses for jumps are stored in a hard coded look up table before the program is run and we jump by referencing the register that contains the index of the address in the lookup table. The maximum branch distance is assumed to be infinite as the lookup table can hold very large numbers.
If there is a jump longer than the available bytes for the lookup tables, then we would do something like a compound jump where we would jump to one address which jumps again to another address.

Addressing Modes

TODO. What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.
We are using both direct and indirect. In some cases, we have a register holding the address being used such as the when jumping. We use indirect addresses for the R-type instructions. For example, add R7 would sum the values stored in R0 and R7, and store it in R0. We also use indirect addressing for jump instructions. For example, a jump instruction would be *jlt #4* which would mean for the program to jump to the address at index 4 of the lookup table. Our I-type instructions are the only ones that use direct addressing. We use we use direct addresses such as *movi #22* which would let the program know to move the number 22 into the register.

4. 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.
A programmer using our architecture should generally think about storing all the values they need into a lookup table before running the program. We only offer 16 registers so they should be moderately conservative about the amount of data they have to store while running the program. These registers are only 8 bits wide as well, giving yet another reason to store some of the values needed before the program is run. Because the ISA is an accumulator, they should be prepared to think about moving many values in and out of registers to be able to manipulate multiple values at a time.

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.
We are unable to copy the instructions from MIPS or ARM ISA because of the difference in the format of the instructions. In MIPS and ARM, their instructions use up to three register operands, one destination register, and two operating registers. In our instructions, we are using an accumulator register which means for our instructions, we are only using one register which all eventually lead to the accumulator register. This does not mean that ARM or MIPS functions and codes would not work in our program, but it means that there would be more instructions because we would need to move information out of the accumulator and into another register or vice versa.

TODO. 4.3 Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?
No, our ALU is only used for our R-type (arithmetic) instructions. We do not allow for addressing offsets; any memory accesses must have the absolute address desired to be accessed stored in a register.

5. 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 Note: this will be completed for Milestone 3, but start thinking about it actively now.

Program 1 Assembly Code

TODO Likewise, Milestone 3.

Program 2 Pseudocode

TODO

Program 2 Assembly Code

TODO

Program 3 Pseudocode

TODO

Program 3 Assembly Code

TODO

ChangeLog from Milestone 1

1. Changed the number of bits to specify the operand register from 5 to 4. We realized that the assignment specifies we can only have 16 registers.
2. Added a lookup table for the program counter in order to store jump addresses. We were using registers to begin with, but this way, jumps are limited to 255 addresses away. With the lookup table, we are not bounded by how far the PC can jump.
3. Added the `cmp` and `jeq` instruction to set the flag if the two registers are equal, and to jump in the case that they are equal.
4. Added the `str` instruction because we realized the testbench read the memory values to see our answer.
5. Added a `jmp` instruction to unconditionally jump to the target when called.
6. Changed from 3 flags to 1 flag. We will have 1 flag which can represent different scenarios depending on what the instruction was that set the flag. `cmp` sets the flag if the result of subtracting Rs from R0 is 0. `sub` sets the flag if the resulting value from subtracting Rs from R0 is negative. `lsl` sets the flag if the bit that was shifted out last was 1.
7. Dedicated R15 to be the status register that will hold the value of the flag. We realized we needed a place to easily store and access the flag.

ChangeLog from Milestone 2

1. Added a right shift logical `rsl` operation.
2. Changed the status register from R15 to R3.