CSE 141L Milestone 1

Esther Xiong, A17043893; Eban Covarrubias, A16743935; Charlie Trinh, A18061636

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.

Esther Xiong
Eban Covarrubias
Charlie Trinh

0. Team

Esther Xiong
Eban Covarrubias
Charlie Trinh

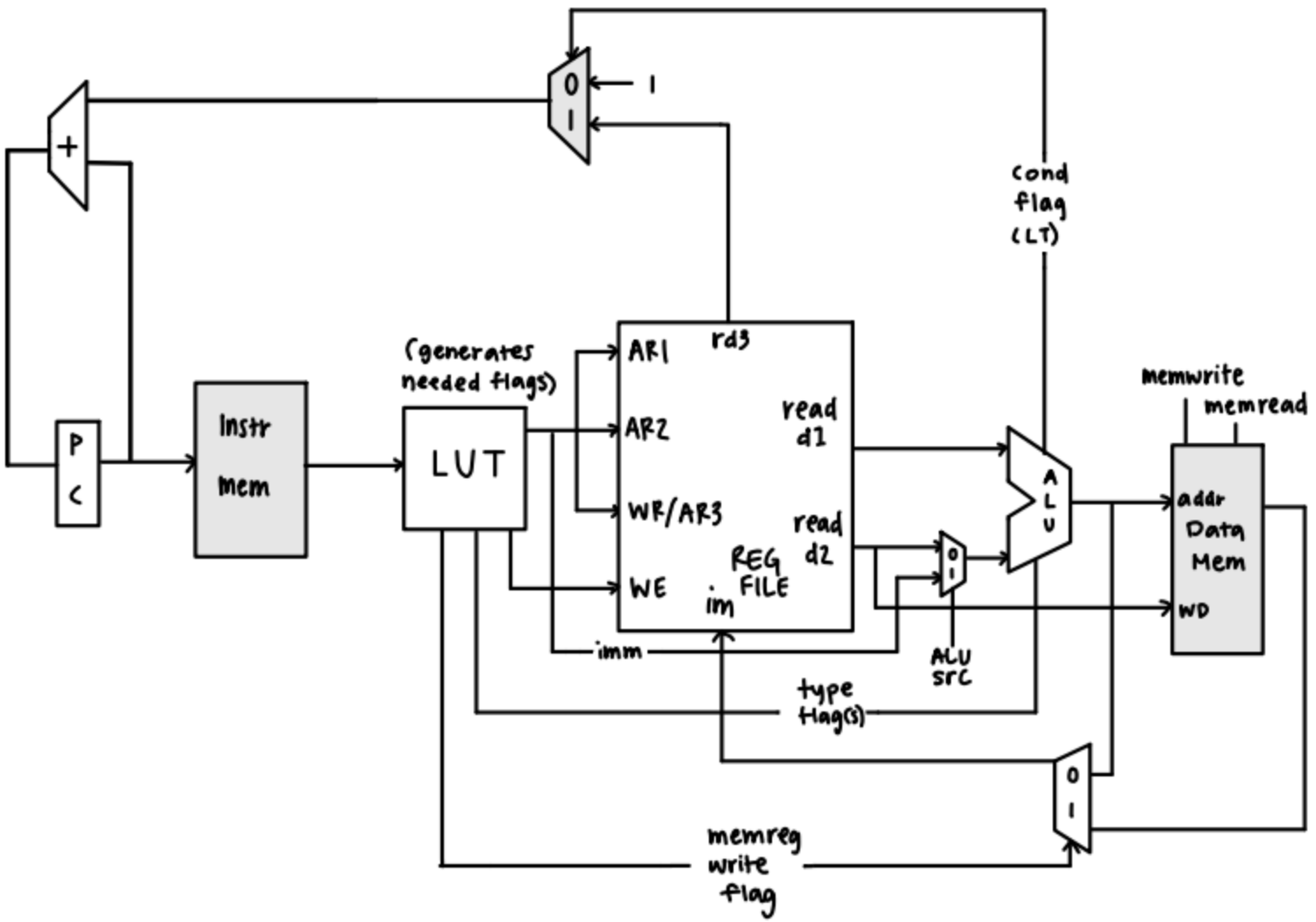
1. Introduction

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.

What we want:
Simplified Mips Architecture
Our goal is to make sure we can support all essential commands while working with a limited number of bits. We want to minimize both the size of our instruction set, and try to keep a short list of available registers. We will limit ourselves to only 4 general use registers, each of which have 8 bits of memory. This is to minimize the number of bits used up on the instructions for referencing registers. We will have a total of 8 operations to pick from, which will take up the first 3 bits of each instruction, using different opcodes. The instructions we will use will be specified lower on this document.

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)


+
0
1
1
PC
Instr
mem
LUT
(generates
needed flags)
AR1
AR2
WR/AR3
WE
rd3
read
d1
read
d2
REG
FILE
im
imm
cond
flag
(LT)
ALU
0
1
ALU
src
memwrite
memread
addr
Data
Mem
WD
type
flag(s)
memreg
write
flag

3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
Mem	3 bits opcode, 2 bits send/destination register, 2 bits address register, 2 bits offset.	str/ldr
B	3 bits opcode, 2 bits destination line register, 2 bits register to compare 1, 2 bits register to compare 2.	blt
R	3 bits opcode, next 6 bits are used for 3 registers. Each register takes 2 bits. We have Register output, register input 1, register input 2.	xor , and
Data Transfer	3 bits opcode, 2 bits for destination register, 4 bits for immediate input.	Mov, shift
Arithmetic	3 bits opcode, 1 bit to denote signed addition or not, 1 bit for carry in, 2 bits for destination register, 2 bits for register holding added value.	SaddTo

Should we reduce types??? Someone look into this

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
Xor = exclusive bitwise or.	R	3 bits for opcode (110), 2 bits for output register(XX), 2 bits for input 1 register(XX), 2 bits for input 2 register(XX).	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 Xor r2, r0, r1 110 10 00 01 #after instruction r2 will hold 0b0011_0010	
And = bitwise and function	R	3 bits for opcode(111), 2 bits for output register(XX), 2 bits for input 1 register(XX), 2 bits for input 2 register(XX).	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 And r2, r0, r1 111 10 00 01 #after instruction r2 will hold 0b0000_0001	
Saddto = signed addition to register	Arithmetic	3 bits opcode(100), 1 bit to denote signed addition or not (X) [1 represents signed, 0 represents unsigned], 1 bit for carry in (X), 2 bits for destination register(XX), 2 bits for register holding value to be added (XX).	#Assume R0 has 0b0001_0001 #Assume R1 has 0b0010_0011 Saddto 0, 1, r0, r1 100 0 1 00 01 #after instruction r0 will hold 0b0011_0101	This example uses unsigned addition with a carry in bit. We will also be able to do signed addition, and carry in bit is optional.

Mov = move instruction copies data from one location to another	Data Transfer	3 bits opcode (010), 2 bits for destination register(XX), 4 bits for immediate input (XXXX). This will allow value assignment from 0 to 15.	Mov r0, 0001 010 00 0001 #after instruction r0 will hold 0b0000_0001	Note that we can only represent up to the number 15 in binary. If we want other number we must use this function in combination with both saddto and shift.
Shift = Shift some register value a specified number of bits.	Data Transfer	3 bits opcode(011), 2 bits destination register(XX), 4 bits signed shift amount.	#assume r0 hold 0b0000_1010 Shift r0, 0001 #after instruction r0 will hold 0b0000_0101.	Note that this shift is logical shift only, we can change the direction by using a negative value for the shift offset value. Positive is logical right shift, negative is logical left shift.
blt= branch if less than, relative branch	B	3 bits opcode(101), 2 bits for register holding destination address register (XX), 2 bits register with value 1 (XX), 2 bits register with value 2(XX)	#assume r0 holds 0b1000_0000 #assume r1 holds 0b0000_0001 #assume r2 holds 0b0000_0011 Blt r0, r1, r2 #because r1 is less than r2 (1 < 3) we will move our pc (program counter) -128 as specified by r0. So if we are on line 128 our branch moves the pc to line 0.	In the case that we move to far in memory, aka out of range, we will simply cap the pc movement. For example, if we are on line 1 and try to move -128 lines, we will stop at line 0.

Str = store value in memory	mem	3 bits opcode(000), 2 bits register with data to send(XX), 2 bits register with memory address(XX), 2 bits for offset.	#assume r0 hold 0b0000_0011 #assume r1 hold 0b0001_0000 Str r0, r1, 00 #after this command, at memory address 16 we will store the byte 0b0000_0011 as specified by r0.	Note that the offset in this example is 0. The offset is added to the specified memory address to store data at. We can only store one byte at a time.
ldr= load value from memory	mem	3 bits opcode(001), 2 bits register to receive data (XX), 2 bits register with memory address(XX), 2 bits for offset.	#assume memory address 16 holds a value of 0b'0011_1100 #assume r1 holds the value 15, or 0b0000_1111 Ldr r0, r1, 01 #after this r0 will hold 0b'0011_1100	Note that we used the offset to increment r1 from 15 to 16 in order to load from address 16. The reason I did this is because in practice we probably will use a mov statement to get the correct address into a register, and mov only has the range 0-15 which makes the offset useful for reducing lines of instructions.

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?
We will have 4 registers, represented by 00, 01, 10, 11. Each of these registers will be general purpose 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 are only supporting one branch statement, which is branch if less than, (blt) The max branch distance we can support is anywhere in the range [-128, 127]. Whatever number is held in the first argument register is how far from our current program position we will move. In order to accommodate larger jumps, we must use multiple blt statements that jump to eachother.

Addressing Modes

TODO. What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.
We use direct addressing for our load and store operations. The addresses are represented by the 8 bits in our second register argument, which gives a range of [0, 256]. The addresses are calculated by using a standard binary representation. We also add the offset argument value, which has a range of [0, 3].

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.

Due to having very limited registers, (only 4) the programmer should use load and store to save values for future use. The registers will likely be changing values very frequently for larger more complex programs. We also have limited operations since our instruction bit string is so short, which means getting specific values into registers using mov can be challenging. In order to get larger values (eg over 15) into a register, the user must use the saddto operation or shift if they are able to reach this value by multiplying by multiples of 2, or a combination of the two operations.

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 not able to copy due to the limited number of instruction bits. We are dealing with this by reducing both the amount of operations that we can use, having more general use operations, and reducing the number of registers to only 4. This current design also doesn't use any bits for type specification. Which means, we likely will use a lookup table to generate these bits based on our opcodes.

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

```
#include <stdio.h>
#include <stdint.h>
#include <limits.h> // For INT_MAX

// Assuming `n` is defined somewhere and points to an array of integers.
```

```
extern int n;
extern int array[];

// Function to calculate the minimum and maximum differences
void calculate_min_max_diff(int* array, int n, uint8_t* output) {
    int min_diff = INT_MAX; // Initialize min_diff to the maximum 16-bit signed value
    int max_diff = 0;       // Initialize max_diff to 0

    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            int diff = array[i] - array[j];
            if (diff < 0) {
                diff = -diff; // Take the absolute value
            }

            if (diff < min_diff) {
                min_diff = diff; // Update minimum difference
            }
            if (diff > max_diff) {
                max_diff = diff; // Update maximum difference
            }
        }
    }

    // Store results in the output memory locations
    output[0] = (min_diff >> 8) & 0xFF; // Store upper 8 bits of min_diff
    output[1] = min_diff & 0xFF;        // Store lower 8 bits of min_diff
    output[2] = (max_diff >> 8) & 0xFF; // Store upper 8 bits of max_diff
    output[3] = max_diff & 0xFF;        // Store lower 8 bits of max_diff
}
```

Program 2 Assembly Code

TODO

```
ldr r0, =0x40 // load address of n into r0
ldr r1, [r0] // load n into r1
sub r1, r1, #1 // n-1 for the loop counter for outer loop


//initalize min and max
mov r2, #0x7FFF // max 16 bit singed value for min diff
mov r3, #0 // 0 for max diff

outer_loop:
    mov r4, #0 // initalize inner loop
    add r4, r4, r0 // adds the base address to r4

inner_loop:
    ldr r6, [r4, r4, lsl #1] // loads value at r4 into r6
    add r5, r4, #1
    ldr r7, [r4, r5, lsl #1] // load next value into r7

    sub r8, r6, r7 //calcuates the difference
    cmp r8, #0
    bge positive // is r8 >= 0 branch to positive

** we need to make an absolute value command **

negative:
    abs r8 // takes the absolute value if negative

positive:
```

```
//compares the current difference to see if it's smaller than the current minimum
    cmp r8, r2
    bge next // if r8 >= r2, branch to next
    mov r2, r8 // update minimum

next:
//compares the current difference to see if it's larger than the current maximum
    cmp r8, r3
    ble cont // if r8 <= r2, branch to cont
    mov r3, r8 //updates the max

cont:
    add r4, r4, #1 // increment inner loop index
    cmp r4, r1
    blt inner_loop // if r4 < n-1, continue inner loop

    sub r1, r1, #1 // decrement outer loop index
    cmp r1, #0
    bge outer_loop // if r1 >= 0, continue outer loop

    //storing results

    ldr r0, =0x42 // load address of memory location 66

    mov r10, r2, lsr #8 // extract the upper 8 bits of the minimum difference
    strb r10, [r0] //store upper 8 bits to memory location 66
    add r0, r0, #1 // moves to the next memory location 67
    mov r10, r2, lsl #24
    mov r10, r10, lsr #24 //extract the lower 8 bits of minimum difference
    strb r10, [r0] //stores the lower 8 bits to memory location 67

    add r0, r0, #1 // moves to next memory location 68
```

```
mov r10, r3, lsr #8 // extract the upper 8 bits of the maximum difference
strb r10, [r0] //store upper 8 bits to memory location 68
add r0, r0, #1 // moves to the next memory location 69
mov r10, r3, lsl #24
mov r10, r10, lsr #24 //extract the lower 8 bits of maximum difference
strb r10, [r0] //stores the lower 8 bits to memory location 69
```

Program 3 Pseudocode

TODO

Program 3 Assembly Code

TODO