CSE 141L Milestone 1

Ryan Lee, A17045234

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.

Ryan Lee

0. Team

Ryan Lee

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.

This document outlines the Eepy architecture. The architecture made me feel eepy every time I had to work on it, and is named thusly. It is a load-store architecture that utilizes accumulators to facilitate the use of extremely small datapaths. So, it is an accumulator-load-store hybrid architecture.

The goal of this architecture is to provide a robust instruction set for operating on 16-bit data types using only 8-bit registers, per the assignment limitations. For example, there are instructions for adding carries, and shifting in overflowed shift values. It also strives to maintain a familiar perspective on the load store architecture despite most of the instruction set being consumed by the float-specific instructions, hence the frame buffer and the load/store instructions.

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)

We will adhere to a simple 5-stage pipeline architecture, with the classic Fetch, Decode, Execute, Memaccess, and Write stages. We will have bypass and stall logic to avoid data hazards. I did not have time to make a diagram :(

3. Machine Specification

Instruction formats

The below table is my first draft of instructions. All instructions contain a register as the left operand; this is usually the destination operand. The “R” type stands for “register,” as the right operand is a register. The “I” stands for “immediate” because the right operand is an immediate value.

TYPE	**FORMAT**	**CORRESPONDING INSTRUCTIONS**
MVI	0, 8 bits imm	mvi
MOV	100, 3 bits op reg 1, 3 bits op reg 2	mov
MIM	101, 1 bit opcode, 5 bits imm	ldi, sti
BOP	11, 4 bits opcode (excluding 1111), 3 bits op	Adi, adr, rsr, lsr, rsi, lsi, bo, ban, lod, str, sac, sfp, skp, nzs
UOP	111111, 3 bits opcode	bne, lne, nop, sho, adc, jmp, ext

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
and = logical and	R	1 bit type (0) bits opcode (010), 1 bit funct (1), 1 bit operand register (X), 1 bit operand register (X), 2 bit destination register (XX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `and R0, R1, R2` ⇔ `0_010_1_0_1_10` # after and instruction, R2 now holds `0b0001_0000`	This is a completely bogus example, since this implies that there are only 2 possible operand registers and 4 possible destination registers. Mention special things like implied destination register (i.e. stack) or special notes here.

The following bullet points denote a tree. The inner nodes indicate prefixes, and the leaves indicate instructions.

- 0 (1 instruction)
    - MVI: Move 8-bit immediate into accumulator register
- 1 (16 instructions, 8 bits left)
    - 0 (7 bits left)
        - 0 (6 bits left -> 3 bits op1, 3 bits op2)
            - MOV: Move
        - 1 (6 bits left)
            - 0 (5 bits left)
                - LDI: store accumulator value into address represented by 5-bit immediate (faster than storing address in a register, at expense of limited range)

        - 1
            - STI: store accumulator value into address respresented by 5-bit immediate (faster than storing address in a register, at expense of limited range)
- 1 (7 bits remaining)
    - 0000-1110 (3 bits remaining -> accum as op1, 3 bits as op2)
        1. ADI: Add immediate
        2. ADR: Add register
        3. RSR: Rshift (will shift left if register is negative)
        4. LSR: LShift (will shift right if register is negative)
        5. RSI: Rshift immediate
        6. LSI: Lshift immediate
        7. BOR: bitwise Or
        8. BAN: Bitwise and
        9. LOD: Load from address in reg into accum
        10. STR: Store value in accum into address in reg
        11. SAC: Set accumulator (does not depend on current accum)
        12. SFP: Set frame pointer to value in register
        13. SKP: guarantee a skip forward by 1-8 instructions
        14. NZS: Conditional skip forward by an immediate value between 1-8 (if accum is nonzero)
    1. 1111 (3 bits remaining)
        1. 000-111 (0 bits remaining -> accum is the only operand)
            1. BNE: bitwise negation
            2. LNE: Logical negation
            3. NOP: no-op
            4. sho: Shift overflow
            5. ADC: Add carry
            6. JMP: jump using r6 as high address bits and r7 as low address bits
            7. EXT: exit the program

Internal Operands

This architecture supports 8 general-purpose registers, numbered r0 through r7. As can be seen from the operations section, one of these registers can be set to be the accumulator for subsequent operations using the `sac` instruction, and will be used as the first operand in most binary operations as well as the destination register. r6 and r7 are special because they are used as the registers for the `jmp` instruction; this instruction will concatenate the bits in r6 and r7 to form a 16-bit address which is then moved into the instruction counter. r6 becomes the high bits, r7 becomes the low bits.

R7 is even more special because it is the register to which immediates are written to. This fact can be exploited to generate masks or jump addresses and utilize them without having to switch the accumulator, saving instructions.

There are a few other special-purpose registers that cannot be directly accessed/addressed. However, the data they hold impacts the outcome of several key instructions. One is the accumulator selector register, denoted asr. Obviously, it points to which register is the current accumulator.

Another is the shift overflow register, denoted the sor. This is used when the programmer has just shifted bits out of a register, and wants to shift bits into another one. The sho instruction (for shift in overflow) causes the machine to read the shift overflow register and shifts in the data it stored into another register. This register is useful because we are working with 16-bit data types and oftentimes want to shift across registers, as though two registers were one.

* I just realized that the shift overflow register, being an 8-bit register, can only store overflows of up to 8 bits. So if we want to shift more than that, we will need multiple instructions or a loop! I did not write my assembly with that in mind...

The addition carry register is similar, and is specifically used to store the carry bit if a carry overflow occurs. This is useful for two’s complement addition across multiple registers.

The frame base register is used for the ldi and sti instructions. When these instructions are running, the machine will read the frame base register’s value and add the immediate passed to ldi/sti. The resulting value is the address from which the machine will load a value if ldi is run, or to which the machine will store the current accumulator’s value if sti is run.

Finally, the program counter determines the next instruction to be fetched.

Control Flow (branches)

There are 3 instructions useful for branching. 2 of them, skp (for “skip”) and nzs for (“nonzero skip”) are relative. Skip takes an immediate value between 1 and 8, and always skips that many instructions. Nonzero skip is similar, but will only skip if the accumulator is nonzero. Nonzero skip is the main tool for conditional branching; programmers should compute the decision for conditional branching in our accumulator first, then execute the nonzero skip once this decision is reached.

The third kind of branching instruction is the jmp instruction. The instruction itself takes no operands, but implicitly uses r6 and r7 to construct the jump address is explained before.

Addressing Modes

TODO. What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

Memory is addressed indirectly. The instructions to load from memory are lod (for “load”) and ldi (for “load using immediate”). Lod takes any gp register as an operand and interprets the value as an address. The machine will then load the memory at that address into the current accumulator. Ldi obtains an address by adding the current frame base register with the immediate encoded in the instruction. Then ldi loads the current accumulator’s value into that address in memory

Sti and str work similarly. Str uses the address stored in a register and stores the current accumulators value at that address. sti uses the address obtained from adding the frame base pointer and the immediate.

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.

This machine is a load-store processor with the ability to choose “accumulators” to operate on (although they are not accumulators in their strictest sense). It is best to think of specific addresses in memory should be mapped to specific variables. Registers take on those variables' values when they load those addresses. Consider the current accumulator as the variable you are currently operating on. Operating on the accumulator typically involves another register, and is akin to evaluating an operation on two tokens (the two tokens being the accumulator and the other register). Assignment is akin to storing a value in a variable.

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.

Many instructions are similar to the ones from the ARM ISA, but naturally have been modified to account for the reduced width of instructions. The main means of overcoming this limitation was by storing one of the operands for most operations within the machine's state, thus removing one instruction encoding from almost every instruction I needed. I could have moved the encoding of the second operand in most binary operations into the machine state as well, but I wanted to preserve some performance, as often I will want to utilize many different registers as the second operand in most instructions and did not want to have to write a "set operand 2" instruction every time I needed to do so. I felt the bits afforded by moving one register to the machine state was enough.

The other means of improving the expressive power of the instruction bits was by using variable-length opcodes. This allowed me to pick opcode lengths that afforded many opcodes, while leaving the appropriate number of bits for an immediate or a register. On one extreme, I needed 8 bits of immediate (the MVI instruction, or "move immediate" instruction). Therefore, I could not even afford to encode a register, and had to pick a default one (r7). On the other extreme are the unary operations like bitwise negation (BNE), whose opcode consumes the entire 9 bits because no operands need to be encoded. In between are the binary operations, which usually have 6 bits of opcode and 3 bits for a single operand (the other operand being encoded as the accumulator), allowing a comfortable number of instructions.

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

```
# int to float
TODO    Note: this will be completed for Milestone 3, but start thinking about it actively now.
int_to_float(in):
  # f16 is SEEEEEMMMMMMMMMM
  # Sign bit
  out[16];
  if in == 0:
    out = 0
    ack
  out[15] = in[15];
  magnitude = in[15] ? -in : in;
  exp_consume = magnitude
  exponent = -1;
  while exp_consume is not 0:
    exp_consume = exp_consume >> 1
    exponent += 1;
  mantissa = magnitude << (10 - exponent)
  exponent = exponent + 15


  out[14:10] = exponent[4:0]
  out[9:0] = mantissa[15:];
```

Program 1 Assembly Code

TODO   Likewise, Milestone 3.

```
// high in is stored at mem_core [0]
// low in is stored at mem_core [1]

#define INLO 0
#define INHI 1
#define OUTLO 2
#define OUTHI 3
#define SIGN 4
#define EXP 5
#define MANTLO 6
#define MANTHI 7
#define UNSIGNLO 8
#define UNSIGNLO 9

// load integer value for conversion
mvi 0        # set frame pointer to address 0
sfp r7
sac r2       # put low bits of input into r2
```

```
ldi INLO

sac r0          # set accumulator to r0
ldi INHI        # load high input bits into r0 from mem
mov r0 r1         # copy high input bits into r1. We will need that later.

// determine sign
mvi 0x80          # mask only the high bits
ban r7


// decide whether to skip sign flip
nzs 4
mvi .get_exp hi
mov r6 r7
mvi .get_exp lo
jmp

// sign was negative. get positive complement
// simultaneously store unsigned int, as will come in handy later.
sac r2
bne
adi 1
sti UNSIGNLO
sac r1
bne
adc
sti UNSIGNHI

// at this moment we have the unsigned int. if it is 0, the original input was
// 0 and we should exist right away, as the remaining code requires the input
// to be nonzero
```

```
sac r3
mov r1 r3
bor r2 r3
nzs 4
sac r1  # if r1 and r2 were nonzero, we would have skipped this. so r1 = 0
sti OUTHI
sti OUTLO
ext

// Recall: r0 -> sign, {r1, r2} -> unsigned integer input
// now, make r3 -> exponent
// use r4 -> condition for branching (is 0 while {r1, r2} is 0, and nonzero if not)
.get_exp
mvi -1
mov r3 r7

// while loop
.get_exp_l
// shift input integer downward
sac r1
rsi 1
sac r2
sho
sac r3
adi 1

// if {r1, r2} != 0, loop back to beginning of while loop
sac r4
mov r4 r1
bor r2
lne
nzs 4
```

```
mvi .get_exp_l hi
mov r6 r7
mvi .get_exp_l lo
jmp

.get_mant
// The mantissa is simply the unsigned value's 10 highest elements.
// An easy way to extract it is to exploit the exponent we just got
sac r4
mov r4 r3   # move exponent into r4
bne         # negate it
adi 1
mvi 10      # add 10 to the negated exponent
adr r7
// now, r4 = 10 - exponent. This is the shift amount.

// before we get our mantissa, we have to load the input containing the mantissa back into {r1, r2}.
sac r1
ldi INHI
sac r2
ldi INLO

// move mantissa into place. If r4 is negative, move r2 first. Otherwise, move r1.
sac r5      # let r5 decide whether to shift right or not.
mov r5 r4
mvi 0x80    # mask sign bit
ban r7
nzs 5       # if r5 is nonzero, then r4 (the shift amount) was negative. shift r1 first.
sac r2      # r4 was positive if we reached this instruction. shift r2 first.
lsh r4
sac r1      # continue shift with r1
sho
```

```
skp 4       # skip the next 4 instructions, which are used to shift r1 to the right first.
sac r1      # shift {r1, r2} by r4, which is the amount needed to shift mantissa into place
lsh r4
sac r2      # continue shift with r2
sho

// mask mantissa
mvi 0x03    # prepare mask to extract mantissa only
and r7      # mask mantissa

// Now that we've used the exponent to determine the shift amount
// we have the opportunity to bias it and shift it into place
sac r3
mvi 15
adr r7
lsi 2

// recall: r0 -> sign, {r1, r2} -> mantissa, r3 -> exponent
// since we're at r3, might as well build our high byte there.
bor r0
bor r1

// done!
sti OUTHI
sac r2
sti OUTLO
ext
```

Program 2 Pseudocode

```
float_to_int(in):
   out[16];
```

```
    exponent = in[14:10] - 15
    # ignore numbers less than 1. Works for normalized, denormalized numbers and 0s
    if exponent is negative:
        out = 0
        ack
    left_shift_amount = exponent - 10
    out[14:0] = {1, in[9:0]} << left_shift_amount
    if in[15]:
        out = -out
    ack
```

Program 2 Assembly Code

```
#define INLO 0
#define INHI 1
#define OUTLO 2
#define OUTHI 3
mvi 4         # set frame pointer to address 4
sfp r7
sac r2        # put low bits of input into r2
ldi INLO

// get the exponent
sac r0        # set accumulator to r0
ldi INHI      # get high bits, where exponent is located
mvi 0x3e      # get exponent mask
and r7        # mask exponent
rsi 2         # shift exponent into place
mvi -15
adr r7

// if exponent is negative, exit early
```

```
sac r1          # r1 will store our decision
mov r1 r0       # r0 is the exponent we want to check
mvi 0x80        # mask the sign bit out
bad r7
lne             # if sign bit is 1 (negative), do NOT skip. So, NOT it.
nzs 4
sti OUTLO
sti OUTHI
ext

// exponent is not negative. We have an actual nonzero integer on our hands
// Compute the amount we have to shift the mantissa with leading 1 to obtain
// final answer
sac r0
mvi -10
adr r7

//get the mantissa, stored in {r2, r3}
sac r3          # do low mantissa first; want to operate on high mantissa after
ldi INLO
sac r2          # now do high mantissa
ldi INHI
mov r4 r2       # we will need the high mantissa again later to obtain the sign
mvi 0x03        # mask non-mantissa bits
ban r7

// stick a one in front of the mantissa
mvi 0x04
bor r7

// shift the mantissa with the 1 until it becomes the integer value
// recall the shift amount was stored in r0
```

```
sac r5      # let r5 decide whether we will shift {r2, r3} right or not.
mov r5 r0
mvi 0x80    # mask sign bit
ban r7
nzs 5       # if r5 is nonzero, then r0 (the shift amount) was negative. shift r1 first.
sac r3      # r0 was positive if we reached this instruction. shift r2 first.
lsh r0
sac r2      # continue shift with r1
sho
skp 4       # skip the next 4 instructions, which are used to shift r1 to the right first.
sac r2      # shift {r2, r3} by r4, which is the amount needed to shift mantissa into place
lsh r0
sac r3      # continue shift with r2
sho

// if the float was signed, negate.
// utilize r4, which was had the high input bits moved in earlier.
sac r4
mvi 0x80    # mask sign bit
ban r7
lne         # if sign is negative (1), DONT skip. so we have to NOT r4.
nzs 4
sac r3
bne
adi 1
sac r2
bne
adc

// Final integer is stored in {r2, r3}.
sac r2
sti OUTHI
```

```
sac r3
sti OUTLO
ext
```

Program 3 Pseudocode

```
float_add(f1, f2):
    out[16];
    subtracting = f1[15] ^ f2[15]
    # get difference between exponent
    exp1 = f1[14:10] - 15
    if f1[14:9] == 1:
        exp1 += 1
    exp2 = f2[14:10] - 15
    if f2[14:9] == 1:
        exp2 += 1
    exp_diff = f1[14:10] - f2[14:10]

    sign = f1[15]
    if exp_diff is negative:
        exp_diff = -expdiff
        max_mant = {1, f2}
        min_mant = {1, f1}
        exp = exp2
        sign = !sign
    else:
        max_mant = f1
        min_mant = f2
        exp = exp1

    if exp_diff > 11:
        out = {sign, exp, max_mant}
        ack

    min_mant = min_mant >> exp_diff
    if subtracting:
        sum = max_mant - min_mant
        if sum < 0:
            sum = -sum  # in the case that exp_diff == 0 and we didn't catch that min_mant was actually larger
            sign = !sign
        # shift down
        while sum[15:10] != 1:
            sum << 1
    else:
        sum = max_mant + min_mant

        if sum[11]:
            exp += 1
            sum >> 1
    result = {sign[0], exp[4:0], sum[9:0]}
```

Program 3 Assembly Code

```
// I'll worry about this later lol
// It looks like a beast
// Wish I had a compiler to slay it
```