CONSIDER THIS DOCUMENT OUTDATED. ALL RELEVANT INFORMATION IS OLD AND OUTDATED; SEE GIT REPO FOR UPDATED INFORMATION.

Milestone 1: Meet “Frankie”

Conventions:

- Comparators put the result of their comparison in the accumulator: 0 for false, 1 for true.
- All operations that manipulate the stack automatically move the stack pointer.
- Spek (peek on the stack) takes an immediate, which is the number of words, not bytes, to move away from the stack pointer when peeking.
- All general cases, or most commonly-occuring cases, have 0 as the flag bit. All special cases have 1 for the flag bit.

Procedure Calling Conventions:

- A function’s return value is put into the accumulators. No additional return values are allowed.
- Ra goes on the stack when you call a function.
- A function’s args go into the backup accumulator, and if there is more than one arg push them to the stack after you push ra to the stack.
- When a function returns, it shouldn’t have a stack frame anymore, so caller must back up accumulator if necessary; if the callee backed it up, it would just get erased upon return.
- Caller should also back up the return address onto the stack.
- Arguments are placed onto the stack in the order listed. If a function is called, foo(a, b), then the caller will put the arguments onto the stack in the order ‘a, b’, and foo will take them off the stack in the order ‘b, a’.

Special Registers:

- Accumulator
- Compare
- Return Address
- Stack Pointer
- Program Counter

Instruction Types

A flag bit decides whether a command applies to both accumulators or to the current accumulator and an immediate

Note: Op code is listed next to the instruction in the instruction list.

Immediate is a binary number.

Generic (G) type instruction-- 1 bit flag, 5 bits for OP code, 8 for immediate.

1: Flag bit	5: OP Code	8: Immediate

Instructions:

put:

- Aput -- G -- 00000 -- (pronounced apud) put in accumulator
    - ML = opcode + 8-bit signed immediate
- Sput -- G -- 00001 -- (pronounced spud) put in stack
    - ML = opcode + 8-bit signed immediate

arithmetic:

- Aadd -- G -- 00010 -- add an immediate value to the value in the accumulator
    - ML = opcode + 8-bit immediate (signed/unsigned?)
- Asub -- G -- 00011 -- subtract an immediate value from the value in the accumulator
    - ML = opcode + 8-bit immediate (signed/unsigned?)

stack manipulation:

- Spek -- G -- 00100 -- “stack peek”; get a value from stack without removing it; takes an unsigned immediate that describes how far back on the stack to peek
    - ML = opcode + 8-bit unsigned immediate
- Spop -- S -- 00101 -- pop value from the stack
    - ML = opcode
- Rpop -- S -- 00110 -- pop value at top of stack into ra

jumps:

- Jimm -- G -- 00111 -- jump a number of words, pc-relative

    - ML = opcode + 8-bit signed immediate
- Jacc -- S -- 01000 -- jump to address stored in accumulator
    - ML = opcode
- Jcmp -- G -- 01001 -- jump if value in comp register is 1
- Jret -- S -- 01010 -- jump to ra
- Jfnc -- S -- 01011 -- jump to a function and set ra to pc+2

compare:

- Cequ -- G -- 01100 -- puts 1 into the comp reg if the value in the accumulator equals the immediate value
    - ML = opcode + 8-bit signed immediate
- Cles -- G -- 01101 -- puts 1 into the comp reg if the value in the accumulator is less than the immediate value
    - ML = opcode + 8-bit signed immediate
- Cgre -- G -- 01110 -- puts 1 into the comp reg if the value in the accumulator is greater than the immediate value
    - ML = opcode + 8-bit signed immediate

logical:

- Lorr -- G -- 01111 -- logical or
    - ML = opcode + 8-bit binary number
- Land -- G -- 10000 -- logical and
    - ML = opcode + 8-bit binary number
- Shfl -- G -- 10001 -- shift left
    - ML = opcode + 8-bit unsigned immediate
- Shfr -- G -- 10010 -- shift right
    - ML = opcode + 8-bit unsigned immediate

mem management:

- Load -- S -- 10011 -- load from memory
    - ML = opcode
- Stor -- S -- 10100 -- store into memory
    - ML = opcode

backup:

- Bkac -- S -- 10101 -- backup accumulator
    - ML = opcode

- Bkra -- S -- 10110 -- backup ra
    - ML = opcode

swap:

Swap -- S -- 10111 -- swap values in accumulators

<u>Note to User:</u>

Our processor is based on an Accumulator with elements of Stack design (A “Frankenstein’s Monster” CPU of sorts, hence the nickname Frankie). There are two accumulators; one main accumulator register in which most of the computation occurs, and a backup accumulator that you can use to swap values to and from the main. Along with the main accumulator register and backup register, there is a small register file for some commonly used values that will need to be stored and kept track of, including the return address, stack pointer, program counter, comparator (which stores the result of comparative operations), and address. Other necessary values are stored or backed up to the Stack.

To use our instruction set, a user needs to understand a bit about the assembly language itself. There is currently one instruction type, which has a 1-bit flag bit, 5-bit OP Code, and 8-bit immediate. Our instructions are broken down above into different categories, depending on what they accomplish. To specify whether a command should be done between the two accumulators or the main accumulator and the stack, a 0 (for general cases, in this case the accumulator and stack) or 1 (for the special cases, in this case accumulator with accumulator) follows the command, with the immediate (if applicable) afterwards. Special and calling conventions are specified above.

Assembly Code Fragments

Add

```
aput 0 2
aadd 0 5
# result: 7
```

Subtract

```
aput 0 5
asub 0 2
# result: 3
```

Basic Stack Functions

```
sput 0 7 # Puts 7 onto the stack
spek 0   # Put top value of the stack into the currently selected
accumulator
sput 0 10 # Puts 10 onto the stack
spop      # Pops off the top value (10) and puts it into the current
accumulator
spop      # Pops off the top value (7) and puts it into the current
accumulator
```

Putting a big immediate into the accumulator

```
aput 0 0b01111111 # Upper half
shfl 0 8
lorr 0 0b11111111
# result: 0b0111111111111111 = 32767
```

Logical Operations

```
aput 0 10
lorr 0 0b10110  # result in accumulator 0b00011
Land 0 0b10110 # result: 0b00010
shfl 0 2
shfr 0 1
```

```
    # result: 0b00100 = 4
```

Load from Memory

```
    Load
```

Save to Memory

```
    Stor
```

Procedure to add 2 + 5

```
    add:
        spop # Pops the top value off of the stack in the current accumulator (2)
        swap # Swaps which accumulator is currently being used
        spop # Pops the top value off of the stack (5)
        aadd 1 0 # Adds both of the accumulators together and stores the result in the first accumulator
        jret # Jump back to ra

    main:
        bkra # Back up the return register to stack
        bkac # Back up both of the accumulators to stack
        sput 0 5
        sput 0 2
        jfnc add # Jump to add procedure, sets ra to pc  + 2
        # When the procedure returns, the return value is in the first
        accumulator
```

Sum numbers 1 - 10

```
            aput 0 0
            swap
            aput 0 1
      loop:
            cgre 0 10
            jcmp 0 exit
            swap
            aadd 1 0 # Add both accumulators and store in first
            swap
            aadd 0 1 # Increment accumulator by 1
            jimm -12
      exit:
```

Old Assembly Code Fragments

Add

```
  aput 2    # put 2 into the accumulator
  aadd 5    # add 5 to the value in the accumulator
  # end result: 7
```

Subtract

```
  aput 5    # put 5 into accumulator
  asub 2    # subtract 2 from value in the accumulator
  # end result: 3
```

Basic stack functions

```
    sput 7    # put 7 onto the stack
    spek 0    # put top value of stack into accumulator
    sput 5    # put 5 onto the stack
    spek 1    # put second value of stack (7) into accumulator
    spop    # remove top value of stack (5), put into accumulator
    spop    # remove top value of stack, put into accumulator
    # end result: 7
```

Put 32767 (big immediate) into accumulator

```
    aput 0b01111111    # put upper half of immediate into accumulator
    shfl 8    # shift binary value in accumulator left 8 places
    orrr 0b11111111 # put lower half of immediate into accumulator
    # end result: 0b0111111111111111 = 32767
```

Logical operations

```
    aput 0b01010
    lorr 0b10110    # after this, 0b00011 is in accumulator
    land 0b10110    # after this, 0b01010 is in accumulator
    shfl 2          # after this, 0b01000 is in accumulator
    shfr 1          # after this, 0b00100 is in accumulator
    # end result: 0b00100 = 4
```

Load from memory

    STILL HAVEN'T DECIDED ON CONVENTIONS

Store in memory

    STILL HAVEN'T DECIDED ON CONVENTIONS

Procedure to add 2+5:

```
    add:
    spop    # pop 5 off stack, into accumulator
    spad    # pop 2 off stack, add it to value in accumulator
    jret    # jump to ra

    main:
    bkra    # back up ra onto the stack
    bkac    # back up ac onto the stack
    sput 2    # prep argument 1; put 2 onto the stack
    sput 5    # prep argument 2; put 5 onto the stack
    jfnc add    # jump to function; sets ra to pc+4

    # jfnc jumps to a function, jret returns from a function
    # spad is also new; it pops a number off the stack and adds it to the
accumulator. It's bad.
```

Add the numbers from 1 to 10 (loops, jumps):

```
    aput 0    # put 0 into accumulator

    aadd 1    # add 1 to value in accumulator; this will be i
    bkac    # back up value in accumulator (i) onto the stack
    bkac    # back up value in accumulator (total) onto the stack

    spek -1    # put i into accumulator
    cles 10    # put 1 into comp register if value in accumulator is less than
10
    aadd 1    # add 1 to i
    bkac    # back up i onto the stack
    spek -1    # put total into accumulator
    sadd    # add value at top of stack to value in accumulator
    bkac    # back up total onto the stack
    jcmp -7    # if 1 in comp register, go back 7 instructions (to spek -1)
```

NOTE: WE NEED TO DEFINE THE CONVENTION FOR THIS

# This method leaves the stack in shambles. It also requires us to create a new instruction (sadd, which is exactly how I feel about this method).

TODO:

1) Update the description of the registers to include the backup accumulator
2) Add any additional procedure call conventions that arise due to the presence of the backup accumulator
   Add variants of commands for operating off the backup accumulator (may necessitate a new type)
3) Add jump types and a few words of explanation for each type
4) Define op codes and any other bits we need
5) Finish gcd(a,b)
6) Write more fragments
7) Translate all assembly into machine code

Notes from meeting:

1) Do we want interrupts/exceptions?
2) How do we want to do I/O? Memory mapped, special inst, or registers?
3) We should be able to add directly off the stack: operate directly on things from memory.
   a) If there are two values somewhere else we want to operate on, we need to put one of them in the accumulator first.
   b) Be careful with code and do a lot of things with just one argument, and memory access won't slow it down too much.
4) Use verilog for everything; don't draw schematics!
5) Our assembler doesn't have to run on our processor; first assemblers are usually built on different machines.
6) First round of documentation ok to have mistakes; we will get extensive feedback, but we need to be pretty clear at least. Maybe use "green sheet" as a model, but no "what's going on" section.
7) Lots of insts that do similar things with minor modifications is best done with funct bit(s), especially if we do multicycle: opens up op codes and makes circuitry easier
8) Multicycle encouraged: *much* faster