TEAM 3V

Maura Coriale
Ben Gothard
Matthew Lyons
Joy Stockwell

"Frankie"

CSSE232-02
Processor Design Document

Table of Contents

Recent Changes

- Added implementation details for all hardware components
- Added slice instead of onion style integration plan
- Added control unit testing plan
- Added the machine code translation to the assembly code fragments

Changes to datapath (not drawn on diagram yet):
  - make ALUOUT an option for pc
  - make ra an option for pc
  - make ALUOUT an option for getting stuff from mem

Changes to accommodate interrupts:
  - Take the interrupt indicating wire to control.
  - Add a wire to the pc mux to jump to the kernel if interrupts are enabled and the interrupt indicator is 1.
  - Take a wire from the overflow detectors to control.
  - Add coprocessor
      - 4 REGS: EPC, cause, backup for mary, and backup for shelley
      - An AND gate with the interrupts enabled bit and the output of the following OR gate
      - An OR gate with a wire from each possible source of exception
          - If any of these is 1 while interrupts is enabled
          - Output is the op of the following MUX
      - MUX that determines whether we run current code or kernel
Control unit:
      - Added section for control unit
      - Added section for control unit testing
      - Added finite state machine for control unit

Meet "Frankie"

Our processor is named Frankie, in reference to Frankenstein's monster. It is primarily accumulator-based, with parts taken from both stack and load-store architecture.

Its main feature is the presence of two accumulators: a main accumulator ("Mary") and a secondary accumulator ("Shelley"). All commands dealing with immediates are handled by Mary. Many commands also have an option of acting on the two accumulators instead; for example, an "aadd" (accumulator add) command could either add an immediate to Mary, or it could add the value in Shelley to Mary. This relationship between the main and secondary accumulators is fundamental to the architecture's design.

Register file

Mary (main accumulator)

This is the main accumulator. Instructions that interact with immediates will interact directly with this. The value in this register is always treated as a signed number in two's complement.

Shelley (secondary accumulator)

This is the secondary accumulator. It can be used as a backup register. It can also be used to perform operations that involve two accumulators. It generally will not interact with immediates. The value in this register is always treated as a signed number in two's complement.

ra (return address register)

This register stores the address that a procedure call will return from using the jret (jump return) instruction. This is set automatically by the jfnc (jump to function) instruction.

pc (program counter register)

This register stores the address of the current instruction. This is set by various jump instruction.

sp (stack pointer register)

This register stores the address of the top of the stack All operations which manipulate the stack implicitly move the stack pointer; as a result, there is no way to set this directly.

comp (comparison result register)

This register stores the result of a comparison instruction (cequ, cles, or cgre), and can only be set by those instructions.

The "Clerval" Instruction Set Architecture

in loving memory of Henry Clerval

There is only one instruction format. It is arranged as follows:

1 flag bit at the start

>This determines whether the instruction will operate on an immediate or on the two accumulators.
>
>If the flag bit is a 0, the command takes an immediate.
>
>If the flag bit is a 1, the command operates on the two accumulators.
>
>Example:
>
>>aadd 0 10 adds 10 to Mary.
>>
>>aadd 1 adds the value in Shelley to Mary.

5 bit op code

>This determines which instruction is performed.

8 bit immediate

>This is always a signed number in two's complement form and will be implicitly sign-extended if it is less than 8 bits.

2 unused bits

>These bits are necessary to make the instruction take up a full two bytes, but they do nothing, and whether they are 0 or 1 has no effect on the instruction itself.

Writing an instruction

All instruction names are 4 characters long. Let "mnem" be the instruction mnemonic and "i" be the immediate; all instructions with flag bit 0 would be written out like the following:

    mnem i

For example, say the user wants to add the immediate value 6 to Mary, the main accumulator. This instruction would be written as follows:

    aadd 6

To set the flag bit to 1, an @ is appended to the end of the mnemonic, like so:

    aadd@

This would perform the alternate aadd instruction, which adds the value of Shelley to Mary.

In cases where the flag bit has no effect, either a 0 or 1 will suffice. In cases where the immediate has no effect, any value will do.
If the flag bit is left blank, it is assumed to be 1.
If the immediate is left blank, it is assumed to be 0.
The mnemonic, of course, cannot be left blank.

Converting an instruction to machine code

To convert an instruction to machine code, the formula is:

    (flag bit) + (op code) + (8-bit immediate) + (00),

where '+' here is assumed to mean "concatenate."
Consider the instruction from the previous example:

    aadd 6

There is no '@' symbol, so the flag bit is 0.
The op code for aadd is 00010.
The 8-bit binary representation of 6 is 00000110.

After concatenating this together, the full machine code instruction
is as follows:
(0) + (00010) + (00000110) + (00) = 0000100000011000

flag	op code	immediate	unused
1	5	8	2

Procedure calling conventions

- When a procedure is called, if the caller requires a backup of the current accumulator values, it is responsible for calling bkac to put them on the stack. It is assumed that the callee is free to overwrite the accumulator values in whatever ways it wishes.

- When a procedure is called, the caller is responsible for backing up the return address register with bkra. The callee is free to overwrite the return address register; it is assumed to be backed up already.

- The first argument to a procedure goes into Mary. The second argument goes into Shelley. Any additional arguments should be put onto the stack after the return address has been backed up.

- After a procedure has concluded, its return value should be put into Mary. If a second return value is needed, it can be put into Shelley. Any additional return values must go onto the stack.

- When a procedure returns, it should no longer have anything remaining on the stack; its stack frame should be completely empty.

Instructions Overview

mnemonic	op code	quick example	quick description
aput	00000	aput 4	sets Mary's value to 4
sput	00001	sput 5	puts 5 on top of the stack
aadd	00010	aadd 4	adds 4 to Mary's value
asub	00011	asub 3	subtracts 3 from Mary's value
spek	00100	spek 0	copies the top value of the stack into Mary
spop	00101	spop 0	pops the top value of the stack into Mary
rpop	00110	rpop	pops the top value of the stack into ra
jimm	00111	jimm LABEL	jumps to the address defined by LABEL
jacc	01000	jacc	jumps to the address denoted by the value in Mary
jcmp	01001	jcmp LABEL	jumps to LABEL if the valutin the comp register is 1
jret	01010	jret	jumps to the value in ra
jfnc	01011	jfnc FOO	jumps to the label FOO and sets ra to pc+2
cequ	01100	cequ 5	sets the value in the comp register to 1 if the value in Mary is equal to 5
cles	01101	cles 6	sets the value in the comp register to 1 if the value in Mary is less than 6

cgre	01110	cgre 2	sets the value in the comp register to 1 if the value in Mary is greater than 2
lorr	01111	lorr 5	sets the value in Mary to the result of a bitwise "or" of its current value and 5
land	10000	land 4	sets the value in Mary to the result of a bitwise "and" of its current value and 4
shfl	10001	shfl 2	shift the value in Mary left 2 bits
shfr	10010	shfr 2	shift the value in Mary right 2 bits
load	10011	load 0x0	loads the value at 0x0 in memory and copies it into Mary
stor	10100	stor 0x0	copies the value in Mary to the address 0x0 in memory
bkac	10101	bkac	copies the value in Mary onto the top of the stack
bkra	10110	bkra	copies the value in ra onto the top of the stack
swap	10111	swap	swaps the values of Mary and Shelley
noop	11000	noop	does nothing and skips to the next instruction

Detailed Instructions Reference

aput -- "accumulator put" -- op: 00000

Flag bit 0: Puts a value into Mary, overwriting her previous
value.
     Example:
        aput 3 # puts 3 into Mary
     This command overwrites Mary's value with the number 3.

Flag bit 1: Puts a value into Shelley, overwriting her previous
value.
     Example:
        aput@ 6 # puts 6 into Shelley
     This command overwrites Shelley's value with the number 6.

sput -- "stack put" -- op: 00001

Puts an immediate value directly on top of the stack.
The flag bit has no effect on sput.
     Example:
        sput 8 # puts 8 on top of the stack
     This command places 8 on top of the memory stack.

aadd -- "accumulator add" -- op: 00010

Flag bit 0: Adds an immediate value to Mary's.
     Example:
        aput 5 # puts 5 into Mary
        aadd 7 # adds 7 to Mary's current value
     After this command is executed, the value in Mary is 12.

Flag bit 1: Adds the value in Shelly to the value in Mary.
Shelley is unaffected.
     Example:
        aput 4 # puts 4 into Mary
        aput@ 2 # puts 2 into Shelley
        aadd@   # adds Shelley's value to Mary's

After this command is executed, the value in Mary is 6, and
the value in Shelley is 2.

asub -- "accumulator sub" -- op: 00011

Flag bit 0: Subtracts an immediate value from the value in Mary.
    Example:
```
        aput 5   # puts 5 into Mary
        asub 7   # subtracts 7 from Mary's current value
```
    After this command is executed, the value in Mary is -2.

Flag bit 1: Subtracts the value in Shelley from the value in
Mary. Shelley is unaffected.
    Example:
```
        aput 4      # puts 4 into Mary
        aput@ 2     # puts 2 into Shelley
        asub@       # subtracts Shelley's value from
                          Mary's
```
    After this command is executed, the value in Mary is 2, and
    the value in Shelley is 2.

spek -- "stack peek" -- op: 00100

Flag bit 0: Copies a value from the stack into Mary. Unlike a
true stack peek, spek can traverse down the stack in 16 bit
increments.
    Example:
```
        sput 5      # put 5 on top of the stack
        sput 7      # put 7 on top of the stack
        spek 1      # copy the second value on the stack into
                    Mary
```
    After this command is executed, Mary's value is 5. The
    stack has a 5 on the bottom and a 7 on top.

Flag bit 1: Copies a value from the stack into Shelley. Unlike a
true stack peek, spek can traverse down the stack in 16 bit
increments.
    Example:
```
        sput 5      # put 5 on top of the stack
        sput 7      # put 7 on top of the stack
        spek@ 0     # copy the second first on the stack into
```

Shelley
        After this command is executed, Shelley's value is 7. The
        stack has a 5 on the bottom and a 7 on top.

spop -- "stack pop" -- op: 00101

Flag bit 0: Moves the top value of the stack into Mary.
        Example:
                sput 2   # put 2 on top of the stack
                spop   # move the value on top of the stack into
                            Mary
        After this command is executed, Mary's value is 2, and the
        stack is empty.

Flag bit 1: Moves the top value of the stack into Shelley.
        Example:
                sput 6   # put 6 on top of the stack
                spop@   # move the value on top of the stack into
                            Shelley
        After this command is executed, Shelley's value is 6, and
        the stack is empty.

rpop -- "ra pop" -- op: 00110

Moves the top value of the stack into ra, the return address
register. If the top of the stack is not a valid address, a
memory exception will likely occur.
The flag bit has no effect on rpop.
        Example:
                bkra   # back up value of ra onto the stack
                rpop   # move the value on top of the stack into
                            ra
        After this command is executed, both ra and the stack are
        the same as they began.

jimm -- "jump immediate" -- op: 00111

Flag bit 0: Set pc to the address specified by the immediate. If the immediate is not a valid address, a memory exception will likely occur.
    Example:
        jimm 0x0   # jump to the address 0x0
    After this command is executed, the value in pc will be 0x0.

Flag bit 1: Add (16*immediate) to the current pc. This effectively moves (1*immediate) instructions forward.
    Example:
        jimm@ -2   # sets pc to pc-32
    After this command is executed, the program will effectively be moved two instructions back.

jacc -- "jump accumulator" -- op: 01000

Flag bit 0: Set pc to the value in Mary. If the value in Mary is not a valid address, a memory exception will likely occur.
    Example:
        aput 0x0   # put 0x0 into Mary
        jacc   # jump to the address in Mary
    After this command is executed, the value in pc will be 0x0.

Flag bit 1: Add (16*Mary's value) to the current pc. This effectively moves (1*Mary's value) instructions forward.
    Example:
        aput -2   # put -2 into Mary
        jacc@   # sets pc to pc-32
    After this command is executed, the program will effectively be moved two instructions back.

jcmp -- "jump compare" -- op: 01001

Flag bit 0: Acts exactly like jimm, but only operates if the value in the comp register is 1; otherwise it does nothing.
    Example:
        aput 5   # put 5 into Mary
        cles 6   # if the value in Mary is less than 6, set
                     the comp register to 1

```
        jcmp 0x0   # jump to the address 0x0 if the value in
                     the comp register is 1
    After this command is executed, the value in pc will be
    0x0.


Flag bit 1: Acts exactly like jimm, but only operates if the
value in the comp register is 1; otherwise it does nothing.
    Example:
        aput 5     # put 5 into Mary
        cles 6     # if the value in Mary is less than 6, set
                     the comp register to 1
        jcmp@ -2   # sets pc to pc-32 if the value in the
                     comp register is 1
    After this command is executed, the program will
    effectively be moved two instructions back.
```

jret -- "jump return" -- op: 01010

```
Sets the pc to the value in ra.
The flag bit has no effect on jret.
    Example:
        jret       # sets pc to ra
    After this command is executed, the program will continue
    execution at ra's position.
```

jfnc -- "jump function" -- op: 01011

```
Flag bit 0: Acts exactly like jimm, but also sets ra to pc+2 so
it can be returned back to with jret.
    Example:
        jfnc 0x0   # jump to the address 0x0, set ra to pc+2
    After this command is executed, the value in pc will be
    0x0, and the value in ra will be (starting pc)+2.


Flag bit 1: Acts exactly like jimm, but also sets ra to pc+2 so
it can be returned back to with jret.
    Example:
        jfnc@ -2   # sets pc to pc-32
    After this command is executed, the program will
    effectively be moved two instructions back, and the value
    in ra will be (starting pc)+2.
```

cequ -- "compare equal" -- op: 01100

Flag bit 0: Compares the supplied immediate to the value in Mary.
If they are equal, it sets the value in the "comp" register to 1.
If they are not, it sets the value in the "comp" register to 0.

```
    Example:
        aput 6      # set the value in Mary to 6
        cequ 6      # sets the value in comp to 1 if Mary's
                      value is equal to 6
    After this command is executed, the value in Mary will be
    6, and the value in comp will be 1.
```

Flag bit 1: Compares Mary's value to Shelley's value. If they are
equal, it sets the value in the "comp" register to 1. If they are
not, it sets the value in the "comp" register to 0.

```
    Example:
        aput 6      # set the value in Mary to 6
        aput@ 6     # set the value in Shelley to 6
        cequ@       # sets the value in comp to 1 if
                      Mary's value is equal to Shelley's
    After this command is executed, the value in Mary will be
    6, the value in Shelley will be 6, and the value in comp
    will be 1.
```

cles -- "compare less" -- op: 01101

Flag bit 0: Compares the supplied immediate to the value in Mary.
If Mary's value is less than the immediate, it sets the value in
the "comp" register to 1. Otherwise it sets the value in the
"comp" register to 0.

```
    Example:
        aput 6      # set the value in Mary to 6
        cles 7      # sets the value in comp to 1 if Mary's
                      value is less than 6
    After this command is executed, the value in Mary will be
    6, and the value in comp will be 1.
```

Flag bit 1: Compares Mary's value to Shelley's value. If Mary's
is less than Shelley's, it sets the value in the "comp" register
to 1. Otherwise it sets the value in the "comp" register to 0.

Example:
```
      aput 6      # set the value in Mary to 6
      aput@ 7     # set the value in Shelley to 6
      cles@       # sets the value in comp to 1 if Mary's
                    value is less than Shelley's
```
After this command is executed, the value in Mary will be 6, the value in Shelley will be 7, and the value in comp will be 1.

cgre -- "compare greater" -- op: 01110

Flag bit 0: Compares the supplied immediate to the value in Mary. If Mary's value is greater than the immediate, it sets the value in the "comp" register to 1. Otherwise it sets the value in the "comp" register to 0.
Example:
```
      aput 6      # set the value in Mary to 6
      cles 5      # sets the value in comp to 1 if Mary's
                    value is less than 5
```
After this command is executed, the value in Mary will be 6, and the value in comp will be 1.

Flag bit 1: Compares Mary's value to Shelley's value. If Mary's is greater than Shelley's, it sets the value in the "comp" register to 1. Otherwise it sets the value in the "comp" register to 0.
Example:
```
      aput 6      # set the value in Mary to 6
      aput@ 5     # set the value in Shelley to 6
      cles@       # sets the value in comp to 1 if Mary's
                    value is greater than Shelley's
```
After this command is executed, the value in Mary will be 6, the value in Shelley will be 5, and the value in comp will be 1.

lorr -- "logical or" -- op: 01111

Flag bit 0: Performs a bitwise "or" between the value in Mary and the supplied immediate, and puts the result in Mary. If necessary, this instruction zero-extends the smaller value.
Example:
```
      aput 4      # sets the value in Mary to 4, or 0b100
      lorr 2      # performs bitwise "or" on the value in
                    Mary and 2, or 0b010
```

After this command is executed, the value in Mary will be
0b110, or 6.

Flag bit 1: Performs a bitwise "or" between the value in Mary and
the value in Shelley, and puts the result in Mary. If necessary,
this instruction zero-extends the smaller value.
    Example:

        aput 4   # sets the value in Mary to 4, or 0b100
        aput@ 1   # sets the value in Shelley to 1, or 0b001
        lorr@   # performs bitwise "or" on the value in
                      Mary and the value in Shelley
    After this command is executed, the value in Mary will be
    0b101, or 5.

land -- "logical and" -- op: 10000

Flag bit 0: Performs a bitwise "and" between the value in Mary
and the supplied immediate, and puts the result in Mary. If
necessary, this instruction zero-extends the smaller value.
    Example:

        aput 4   # sets the value in Mary to 4, or 0b100
        land 2   # performs bitwise "and" on the value in
                      Mary and 2, or 0b010
    After this command is executed, the value in Mary will be
    0b000, or 0.

Flag bit 1: Performs a bitwise "and" between the value in Mary
and the value in Shelley, and puts the result in Mary. If
necessary, this instruction zero-extends the smaller value.
    Example:

        aput 4   # sets the value in Mary to 4, or 0b100
        aput@ 1   # sets the value in Shelley to 1, or 0b001
        land@   # performs bitwise "and" on the value in
                      Mary and the value in Shelley
    After this command is executed, the value in Mary will be
    0b000, or 0.

shfl -- "shift left" -- op: 10001

Flag bit 0: Performs a bitwise left shift on the value in Mary by the number of bits specified by the immediate. This instruction zero-extends from the right.

        Example:
            aput 2   # sets the value in Mary to 2, or 0b010
            shfl 1   # shifts the value in Mary left by 1 bit
        After this command is executed, the value in Mary will be 0b100, or 4.

Flag bit 1: Performs a bitwise left shift on the value in Mary by the number of bits specified in Shelley. This instruction zero-extends from the right.

        Example:
            aput 1   # sets the value in Mary to 1, or 0b001
            aput@ 2   # sets the value in Shelley to 2
            shfl@   # shifts the value in Mary left by the
                          number of bits specified by Shelley
        After this command is executed, the value in Mary will be 0b100, or 4.

shfr -- "shift right" -- op: 10010

Flag bit 0: Performs a bitwise right shift on the value in Mary by the number of bits specified by the immediate. This instruction sign extends from the left.

        Example:
            aput 2   # sets the value in Mary to 2, or 0b010
            shfr 1   # shifts the value in Mary right by 1 bit
        After this command is executed, the value in Mary will be 0b001, or 1.

Flag bit 1: Performs a bitwise right shift on the value in Mary by the number of bits specified in Shelley. This instruction sign extends from the left.

        Example:
            aput 4   # sets the value in Mary to 1, or 0b100
            aput@ 1   # sets the value in Shelley to 1
            shfr@   # shifts the value in Mary right by the
                          number of bits specified by Shelley
        After this command is executed, the value in Mary will be 0b010, or 2.

load -- "load from memory" -- op: 10011

Flag bit 0: Loads the value from memory at the address specified
in the immediate and copies it into Mary. Note that only primary
memory (memory with an address whose first 8 bits are 0) is
accessible through this command; other memory must be accessed
through load@.

        Example:
            load 0x0   # loads the value at address 0x0 in memory
                         and copies it into Mary.
        After this command is executed, the value in Mary will be
        the value at the address 0x0 in memory.

Flag bit 1: Loads the value from memory at the address stored in
Mary and copies it into Mary.

        Example:
            aput 0x0   # sets Mary's value to 0x0
            load@   # loads the value at the address in
                         Mary from memory and copies it into Mary
        After this command is executed, the value in Mary will be
        the value at the address 0x0 in memory.

stor -- "store in memory" -- op: 10100

Flag bit 0: Stores the value in Mary into memory at the address
specified by the immediate.

        Example:
            aput 2   # sets Mary's value to 2
            stor 0x0   # stores the value in Mary at the address
                         0x0 in memory
        After this command is executed, the value in Mary will be
        2, and the value at 0x0 in memory will also be 2.

Flag bit 1: Stores the value in Mary into memory at the address
specified by Shelley.

        Example:
            aput 2   # sets Mary's value to 2
            aput@ 0x0 # sets the value in Shelley to 0x0
            stor@   # stores the value in Mary at the address
                         specified by the value in Shelley

After this command is executed, the value in Mary will be
2, the value in Shelley will be 0x0, and the value at 0x0
in memory will be 2.

bkac -- "back up accumulator" -- op: 10101

Flag bit 0: Copies the value in Mary and places it on top of the
stack.
```
     Example:
          aput 2     # sets Mary's value to 2
          bkac       # copies the value in Mary onto the stack
     After this command is executed, the value in Mary will be
     2, and the value at the top of the stack will also be 2.
```

Flag bit 1: Copies the value in Shelley and places it on top of
the stack.
```
     Example:
          aput@ 3    # sets Shelley's value to 3
          bkac@      # copies the value in Shelley onto the
                       stack
     After this command is executed, the value in Shelley will
     be 3, and the value at the top of the stack will also be 3.
```

bkra -- "back up return address" -- op: 10110

Copies the value in ra and places it on top of the stack.
The flag bit has no effect on bkra.
```
     Example:
          bkra       # copies ra onto the stack
     After this command is executed, the value on top of the
     stack will be whatever ra started as.
```

swap -- "swap the accumulators" -- op: 10111

Swaps the value in Mary with the value in Shelley.
The flag bit has no effect on swap.
```
     Example:
          aput 5     # sets the value in Mary to 5
          aput@ 8    # sets the value in Shelley to 8
```

```
        swap        # swaps the values in Mary and Shelley
    After this command is executed, the value in Mary will be
    8, and the value in Shelley will be 5.
```

noop -- "no operation" -- op: 11111

Empty instruction that does nothing and is always skipped.
The flag bit has no effect on noop.

```
        Example:
            noop        # does nothing
        After this command is executed, nothing has happened!
```

I/O

I/O is done using interrupts. Details will be added later. Need to research how they work, conventions, ways to make them work…

Assembly Code Fragments

Add

aput 2	Put the value 2 into the "Mary" accumulator	00000000000010
aadd 5	Add the immediate value 5 to "Mary"	00001000000101
	"Mary" result: 7	
aput@ 5	Put the value 5 into "Shelley"	10000000000101
aadd@	Add the values of "Mary" and "Shelley"	10001000000000
	"Mary" result: 12 "Shelley" result: 5	

Subtract

aput 5	Put the value 5 into the "Mary" accumulator	00000000000101
asub 2	Subtract the immediate value 2 from "Mary"s value	00001100000010
	"Mary" result: 3	
aput@ 3	Put the value 3 into "Mary"	10000000000011
asub@	Subtract "Shelley"s value from "Mary"	10001100000000
	"Mary" result: 0 "Shelley" result: 3	

Basic Stack Functions

sput 7	Puts 7 onto the stack	000001000000111
spek	Put top value of the stack into the "Mary" accumulator	000100000000000
sput 10	Puts 10 onto the stack	000001000001010
spop	Pops off the top value (10) and puts it into the "Mary" accumulator	000101000000000
spop	Pops off the top value (7) and puts it into the "Mary" accumulator	000101000000000

Putting a big immediate into the accumulator

aput 0b01111111	Put upper half into "Mary"	000000001111111
shfl 8	Shift "Mary"s value left 8 bits	010001000001000
lorr 0b11111111	Or the lower 8 bits into "Mary"	001111111111111
	result: 0b011111111111111 = 32767	

Logical Operations

aput 10	Put 10 into the "Mary" accumulator	000000000001010
lorr 0b10110	Or the value in "Mary" with 10110, result in accumulator 0b00011	001111000010110
land 0b10110	And the value in "Mary" with 10110, result: 0b00010	010000000010110
shfl 2	Shift the value in "Mary" 2 bits left	010001000000010
shfl 1	Shift the value in "Mary" 1 bit right	010010000000001
	result: 0b00100 = 4	

Load from Memory

load 0x0012	Load the value at 0x1001 into "Mary"	01001100010010
aput@ 0x0	Put the value 0 into "Shelley"	10000000000000
load@	Load the value into "Shelley" from the address already stored in "Shelley"	11001100000000

Save to Memory

aput 2	Set "Mary" to the value 2	00000000000010
stor 0x00A2	Store the value in "Mary" into address 0x00A2	01010010100010

Procedure to add 2 + 5

add:		

spop	Pops the top value off of the stack in the "Mary" accumulator (2)	00010100000000
swap	Swaps which accumulator is currently being used	01011100000000
spop	Pops the top value off of the stack (5)	00010100000000
aadd@	Adds both of the accumulators together and stores the result in the "Mary" accumulator	10001000000000
jret	Jump back to ra	00101000000000
main:		
bkra	Back up the return register to stack	01011000000000
bkac	Back up both of the accumulators to stack	01010100000000
sput 5		00000100000101
sput 2		00000100000010
jfnc add	Jump to add procedure, sets ra to pc + 2	00101100000000
	When the procedure returns, the return value is in the "Mary" accumulator	

Sum numbers 1 - 10

aput 0 0	Put 0 into the "Mary" accumulator (total)	00000000000000
aput 1 1	Put 1 into the "Shelley"	10000000000001

	accumulator (i)	
loop:		
cgre 0 10	Compare the "Shelley" accumulator to see if the value is greater than 10	00111000001010
jcmp 0 exit	Jump to exit if "Shelley"s value is greater than 10	00100100001010
swap	Switch accumulator to "Mary"	01011100000000
aadd 1	Add both accumulators and store in "Mary"	00001000000001
swap	Switch back to "Shelley"	01011100000000
aadd 0 1	Increment "Shelley" accumulator by 1	00001000000001
jimm -12	Jump 6 instructions up	
exit:		

Euclid's Algorithm

Address	Code	Comments	Machine Code
0x0000	GCD:	n is a and is on the top of the stack, ra is right behind n, m is b and is in the backup accumulator	01111100000000
0x0002	spop	get n into the main accumulator	00010100000000
0x0004	cequ 1	put 1 in the comp reg if n is equal to 1	00110000000001
0x0006	swap	get ready for the loop or return by putting m into the main accumulator	01011100000000
0x0008	jcmp ENDgcd	if 1 is in the comp reg (n == 1) then jump to the end where we return m	00100100010010
0x000A	LOOPgcd:	This is replaced by a noop command	01111100000000
0x000C	cequ 0	puts 1 into the comp reg if m is 0	00110000000000
0x000E	jcmp ENDswap	if m is 0, the comp reg holds 1, so jump out of the loop and return a = n	00100100010000
0x0010	cles 1	if m (b, in main) is less than n (a, in backup) puts 1 into comp reg, ie: puts 1 into comp reg if a > b	00110100000001
0x0012	jcmp ELSEgcd	jump to the else clause if !(a > b) --> !(n > m)	00100100001101
0x0014	swap	if (a > b) --> (n > m), puts n into main accumulator so we can change its val	01011100000000
0x0016	asub@	subtract the value of the backup accumulator from the value of the main accumulator and store the result in the main accumulator; leaves backup alone, our return value is a = n, so leave it there	10001100000000

		to be the return value	
0x0018	swap	puts b back into the main accumulator so we can compare it with 0 at the beginning of the loop (we're about to go back to its top)	01011100000000
0x001A	jimm LOOPgcd	jump to the end	00011100000100
0x001C	ELSEgcd:	converted into a noop instruction	01111100000000
0x001E	asub@	m is in the main accumulator, n in the backup, subtract the value of the backup accumulator from the value of the main accumulator and store the result in the main accumulator; leaves backup alone, this is m = m - n --> b = b - a	00001100000001
0x0020	jimm LOOPgcd		00011100000100
0x0022	ENDswap:	converted into a noop instruction	01111100000000
0x0024	swap	n was in the backup; now it is in the main to be returned	01011100000000
0x0026	ENDgcd:	Converted into a noop instruction	01111100000000
0x0028	jret	jump to the addr in ra (ie a line in relPrime)	00101000000000
0x002A			
0x002C	main:		01111100000000
0x002E	bkra	Back up the ra on the stack before putting the args on the stack so that GCD can get back to relPrime	01011000000000
0x0030	sput nVal	Put n on the stack, for assembly this is all 0s since this comes from IO	00000100000000
0x0032	aput 2	Put 2 (i.e. mVal) in the accumulator	00000000000000
0x0034	swap	put 2 in the backup accumulator	01011100000000
0x0036	LOOPrp:	converted into a noop instruction	01111100000000
0x0038	jfnc GCD	go to the function in the other file; automatically sets ra to the	00101100000000

		address of this line	
0x003A	cequ 1	checks if return value from GCD (which gets stored in the accumulator upon return) is 1 or not. It puts 1 in the comp reg if so, and puts 0 in the comp reg if not.	00110000000001
0x003C	jcmp ENDrp	continue in the loop unless GCD returned 1 which is to say, break out of the loop if GCD returned 1	00100101000110
0x003E	swap	if GCD didn't return 1, get out of the backup accumulator and into the main accumulator	01011100000000
0x0040	aadd 1	add 1 to m	00000000000001
0x0042	swap	put the new m = mOld + 1 back in the backup accumulator so it is the arg for the next time we call GCD	01011100000000
0x0044	jimm LOOPrp	do the loop again	00011100110110
0x0046	ENDrp:		01111100000000
0x0048	swap	get m into the accumulator because it is the return value, pop until the addr on top of the stack is the address of relPrime's caller	01011100000000
0x004A	rpop	restore ra to the address of whatever called relPrime	00011000000000
0x004C	jret	jump to wherever called relPrime	00101000000000

RTL Table

Arithmetic	Compare	Stack	Jump	Swap	Load/Store
PC = PC + 2 inst = Mem[PC]					
flagbit = inst[15] OPCODE = inst[14, 10] imm = inst[9, 2]					

ALUOUT = mary OP shelly/imm	ALUOUT = mary OP shelly/imm	ALUOUT = sp OP imm	PC = LS(SE(imm)) or imm	ALUOUT = shelly	memVal = Mem[imm]OR Mem[shelley] (only load)
mary = ALUOUT	cmp = ALUOUT	mary/shelley = ALUOUT		shelley = mary mary = ALUOUT	mary = memVal OR Mem[shelley] = mary OR Mem[imm] = mary

RTL Reference by Instruction

```
aput:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
     mary = imm

aput@:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
     shelley = imm

sput:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
     sp = sp + 2
     Mem[sp] = imm

aadd:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
     ALUOUT = mary + imm
     mary = ALUOUT


aadd@:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
```

```
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        ALUOUT = mary + shelley
        mary = ALUOUT


asub:
        pc = pc + 2
        inst = Mem[pc]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        ALUOUT = mary - imm
        mary = ALUOUT


asub@:
        pc = pc + 2
        inst = Mem[pc]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        ALUOUT = mary - shelley
        mary = ALUOUT


spek:
        pc = pc + 2
        inst = Mem[pc]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        ALUOUT = imm*2 + sp
        memVal = Mem[ALUOUT]
        mary = memVal




spop:
        pc = pc + 2
        inst = Mem[PC]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
```

```
        memVal = Mem[sp]
        sp = sp - 2
        mary = memVal


rpop:
        pc = pc + 2
        inst = Mem[PC]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        memVal = mem[sp]
        sp = sp - 2
        reg[ra] = memVal


jimm:
        pc = pc + 2
        inst = Mem[PC]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        PC = imm


jimm@:
        pc = pc + 2
        inst = Mem[PC]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        PC = imm << 4 + PC




jacc:
        pc = pc + 2
        inst = Mem[PC]
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
        PC = mary
```

```
jacc@:
    pc = pc + 2
    inst = Mem[PC]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    imm = inst[9, 2]
    PC = mary << 4 + PC

jcmp:
    pc = pc + 2
    inst = Mem[PC]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    imm = inst[9, 2]
    if reg[cmp] == 1:
        PC = imm

jcmp@:
    pc = pc + 2
    inst = Mem[PC]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    imm = inst[9, 2]
    if reg[cmp] == 1:
        PC = imm << 4

jret:
    pc = pc + 2
    inst = Mem[PC]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    imm = inst[9, 2]
    PC = ra

jfnc:
    pc = pc + 2
```

```
    inst = Mem[PC]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    imm = inst[9, 2]
    ra = PC + 1 word
    PC = imm

jfnc@:
    reg[ra] = PC + 1 word
    ALUOUT = imm << 4
    PC = ALUOUT

cequ:
    aluout = mary - imm
    if (aluout == 0)
     comp = 1
    else
      comp = 0

cequ@:
    aluout = mary - shelley
    if (aluout == 0)
      comp = 1
    else
      comp = 0




cles:
    aluout = mary - imm
    if (aluout < 0)
      comp = 1
    else
      comp = 0

cles@:
    aluout = mary - shelley
    if (aluout < 0)
      comp = 1
    else
```

```
        comp = 0
cgre:
    aluout = mary - imm

    if (aluout > 0)
      comp = 1
    else
      comp = 0

cgre@:
    aluout = mary - shelley

    if (aluout > 0)
      comp = 1
    else
      comp = 0

lorr:
    aluout = mary OR imm
    mary = aluout

lorr@:
    aluout = mary OR shelley
    mary = aluout



land:
    aluout = mary AND imm
    mary = aluout

land@:
    aluout = mary AND shelley
    mary = aluout

shfl:
    shiftout = mary SHIFTLEFT imm
    mary = shiftout

shfl@:
    shiftout = mary SHIFTLEFT shelley
```

```
    mary = shiftout


shfr:
    shiftout = mary SHIFTRIGHT imm
    mary = shiftout


shfr@:
    shiftout = mary SHIFTRIGHT shelley
    mary = shiftout


load:
    pc = pc + 2
    inst = Mem[pc]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    memVal = Mem[imm]
    mary = val


load@:
    pc = pc + 2
    inst = Mem[pc]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    memVal = Mem[shelly]
    mary = val
stor:
    pc = pc + 2
    inst = Mem[pc]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    Mem[imm] = mary


stor@:
    pc = pc + 2
    inst = Mem[pc]
    flagbit = inst[15]
    OPCODE = inst[14, 10]
    Mem[shelly] = mary


bkac:
```

```
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     sp = sp + 2
     mem[sp] = mary


bkac@:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     sp = sp + 2
     mem[sp] = shelly


bkra:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     sp = sp + 2
     mem[sp] = ra


swap:
     pc = pc + 2
     inst = Mem[pc]
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     shelley = mary
     mary = shelley
```

RTL Tests

These rtl tests show the state of the relevant regs before their inst
is run, at each step of the inst, and after the inst is complete. They
say what the state of the regs should be after the inst ends. The
tests were all successful, so, every time, the states afterward were
correct. There is a test of aput and each type of inst.

aput rtl test

pc = 22, mary = 121, shelley = 13
aput 10
result should be: pc = 24, mary = 10, shelley = 13

First block:
      pc = pc + 2
      inst = Mem[pc]
After block:
      pc = 24, mary = 121, shelley = 13
Second block:
      flagbit = inst[15]
      OPCODE = inst[14, 10]
      imm = inst[9, 2]
After block:
      pc = 24, mary = 121, shelley = 13, imm = 10

```
Third block:
      mary = imm
After block:
      pc = 24, mary = 10, shelley = 13
      TEST SUCCESS


stack rtl test

pc = 12, sp = 0, stack empty
sput 4
result should be: pc = 14, sp = 2, stack has 4 on it

First block:
      pc = pc + 2
      inst = Mem[pc]
After block:
      pc = 14, sp = 0, stack empty
Second block:
      flagbit = inst[15]
      OPCODE = inst[14, 10]
      imm = inst[9, 2]
After block:
      pc = 14, sp = 0, stack empty, imm = 4
Third block:
      sp = sp + 2
      Mem[sp] = imm
After block:
      pc = 14, sp = 2, stack has 4 on it
      TEST SUCCESS


arithmetic rtl test

pc = 4, mary = 8, shelley = 3
aadd 12
result should be: pc = 6, mary = 20, shelley = 3

First block:
      pc = pc + 2
      inst = Mem[pc]
After block:
      pc = 6, mary = 8, shelley = 3
Second block:
      flagbit = inst[15]
```

```
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
After block:
        pc = 6, mary = 8, shelley = 3, imm = 12
Third block:
        ALUOUT = mary + imm
After block:
        pc = 6, mary = 8, shelley = 3, ALUOUT = 20
Fourth block:
        mary = ALUOUT
After block:
        pc = 6, mary = 20, shelley = 3
        TEST SUCCESS
```

jump rtl test

```
pc = 4
jimm 10
result should be: pc = 10

First block:
        pc = pc + 2
        inst = Mem[pc]
After block:
        pc = 6
Second block:
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
After block:
        pc = 6, imm = 10
Third block:
        pc = imm
After block:
        pc = 10
        TEST SUCCESS
```

compare rtl test

```
pc = 222, mary = 10, comp = 0
cequ 10
result should be: pc = 224, mary = 10, comp = 1

First block:
        pc = pc + 2
        inst = Mem[pc]
```

```
After block:
     pc = 224, mary = 10, comp = 0
Second block:
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
After block:
     pc = 224, mary = 10, comp = 0, imm = 10
Third block:
     aluout = mary - imm
After block:
     pc = 224, mary = 10, comp = 0, aluout = 0
Fourth block:
     if (aluout == 0)
          comp = 1
     else
          comp = 0
After block:
     pc = 224, mary = 10, comp = 1
     TEST SUCCESS


swap rtl test

pc = 4, mary = 1, shelley = 2
swap
result should be: pc = 6, mary = 2, shelley = 1

First block:
     pc = pc + 2
     inst = Mem[pc]
After block:
     pc = 6, mary = 1, shelley = 2
Second block:
     flagbit = inst[15]
     OPCODE = inst[14, 10]
     imm = inst[9, 2]
After block:
     pc = 6, mary = 1, shelley = 2
Third block:
     ALUOUT = shelly
     shelley = mary
After block:
     pc = 6, mary = 1, shelley = 2, ALUOUT = 2
Fourth block:
```

```
        mary = ALUOUT
After block:
        -
Fifth block:
        mary = B
        shelly = A
After block:
        pc = 6, mary = 2, shelly = 1, A = 1, B = 2
        TEST SUCCESS
```

load/store rtl test

```
pc = 4, mary = 42, shelley = 0, Mem[0] = 24
load@
result should be: pc = 6, mary = 24, shelley = 0, Mem[0] = 24

First block:
        pc = pc + 2
        inst = Mem[pc]
After block:
        pc = 6, mary = 42, shelley = 0, Mem[0] = 24
Second block:
        flagbit = inst[15]
        OPCODE = inst[14, 10]
        imm = inst[9, 2]
After block:
        pc = 6, mary = 42, shelley = 0, Mem[0] = 24
Third block:
        memVal = Mem[0]
After block:
        pc = 6, mary = 42, shelley = 0, Mem[0] = 24, memVal = 24
Fourth block:
        mary = val
After block:
        pc = 6, mary = 42, shelley = 0, Mem[0] = 24, memVal = 24
        TEST SUCCESS
```

States for Each Instruction Type

<table>
<tr><td colspan="2" align="center">Stack</td></tr>
<tr><td>pc = pc + 2</td><td>PCWrite = 0</td></tr>
<tr><td>inst = Mem[pc]</td><td>MemRead = 1</td></tr>
<tr><td>flagbit = inst[15]
OPCODE = inst[14, 10]
imm = inst[9, 2]
A = sp, B = SE(LS(imm))</td><td>RegRead = 1
srcA = sp
srcB = SE(LS(imm))</td></tr>
<tr><td>ALUOUT = A + B</td><td>ALUOP = add</td></tr>
<tr><td>memVal = Mem[ALUOUT)
sp = sp + 2 (only for spop)
mary = memVal</td><td>regWrite = 1
RegData = memory
regDst = Mary
spWrite = 1</td></tr>
</table>

<table>
<tr><td colspan="2" align="center">Compare</td></tr>
<tr><td>pc = pc + 2</td><td>PCWrite = 0</td></tr>
<tr><td>inst = Mem[PC]</td><td>memRead = 1</td></tr>
<tr><td>flagbit = inst[15]
OPCODE = inst[14, 10]
imm = inst[9, 2]
A = Mary
B = shelley (1)   OR   B = imm</td><td>regRead = 1
srcA = mary
srcB = shelley OR srcB = imm</td></tr>
<tr><td>ALUOUT = A - B</td><td>ALUOP = sub</td></tr>
<tr><td>cmp = ALUOUT</td><td>regWrite = 1
RegData = AluOut
regDst = cmp</td></tr>
</table>

<table>
<tr><td align="center">Jump (to function)</td></tr>
</table>

pc = pc + 2	PCWrite = 0
inst = Mem[pc]	memRead = 1
flagbit = inst[15] OPCODE = inst[14, 10] imm = inst[9, 2] OR imm = LS(imm)	
ra = pc	regWrite = 1 RegData = pc regDest = ra
pc = imm	PCWrite = 1

Arithmetic	
pc = pc + 2	PCWrite = 0
inst = Mem[pc]	memRead = 1
flagbit = inst[15] op = inst[14, 10] imm = inst[10, 2]	
A = mary B = shelly OR B = imm	srcA = mary srcB = shelly OR srcB = imm
ALUOUT = A op B	ALUOP = op
mary = ALUOUT	regWrite = 1 regDest = mary regData = ALUOUT

Swap	
pc = pc + 2	PCWrite = 0

`inst = Mem[pc]`	`memRead = 1`

`flagbit = inst[15]`
`op = inst[14, 10]`
`imm = inst[10, 2]`	

`ALUOut = shelley`	`ALUOp = 111`

`mary = ALUOut`	`regWrite = 1`
	`regDest = mary`
	`regData = B`

`shelley = mary`	`regWrite = 1`
	`regDest = shelley`
	`regData = A`

	Stack	A & L	Compare	Jump	Load/ Store	Swap
F	PCWrite = 0 memRead = 1					
D						
E	srcA = sp srcB = imm	srcA = mary srcB =shelley OR srcB = imm		regWrite= 1 srcA = pc srcB = imm	srcA = mary srcB = imm	srcA = mary srcB = shelley OR srcB = imm
M	ALUOP = add	ALUOP = OP	ALUOP = sub	pcWrite = ALUOUT	memSrc = ALUOUT MemRead/ MemWrite = 1	regWrite = 1 regDest = mary regData = B
W	regWrite = 1 regDst = mary regData = Mem spWrite = 1	regWrite = 1 regDst = mary regData = ALUOUT	regWrite= 1 regDst= cmp regData = ALUOUT		regWrite = 1 regDst = mary regData = Mem spWrite = 1	regWrite = 1 regDst = shelley regData = A

State Diagram
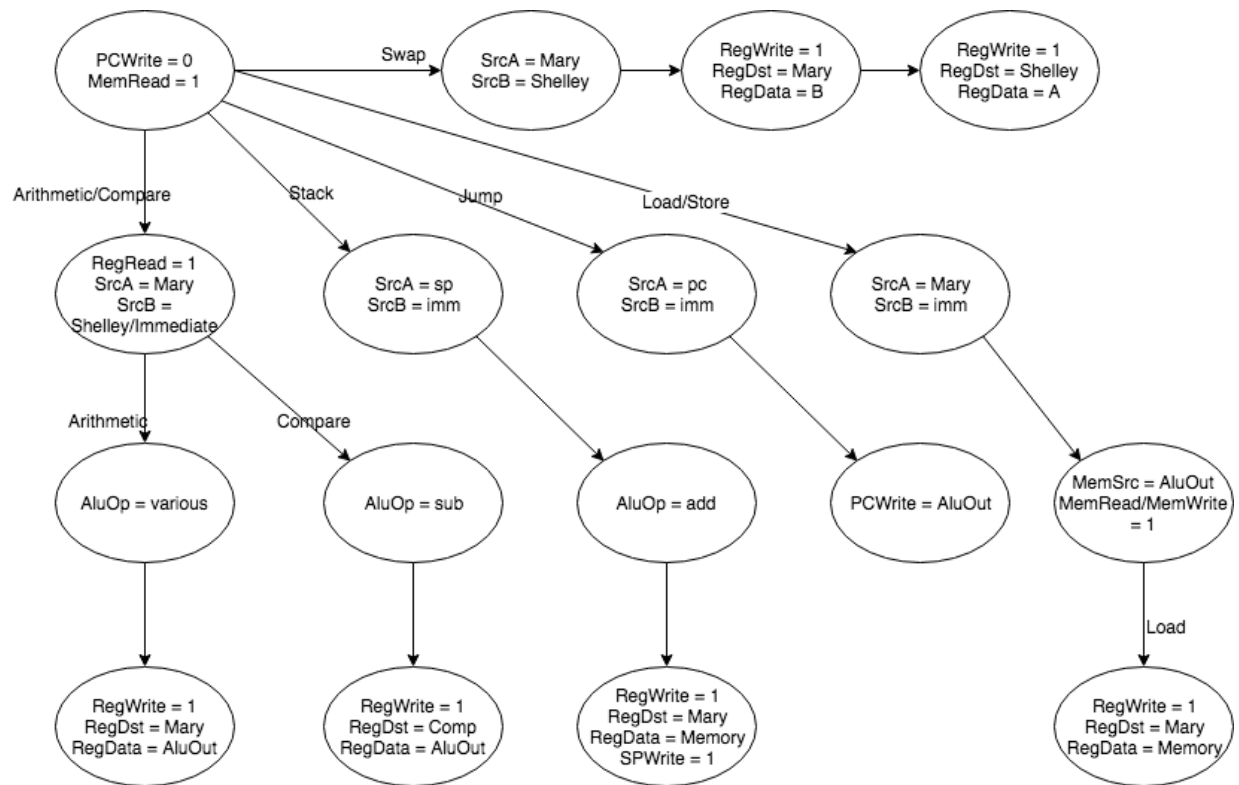
Shopping List

Note: number of bits in each control signal on following list of control signals

- Registers
  - Accumulators: mary and shelly
  - Reg to store return addr of current function: RA
  - Reg to hold result of comparisons: comp
  - Intermediate registers to store data across cycles: PC, SP, A, B, AluOut, Inst, and memVal (retrieved from memory).
  - Input: RegA and RegB, which determine which two registers to read. Both 2 bits. Will usually be Mary and Shelley.
  - Output: ValA and ValB, the values of the two registers specified by RegA and RegB. Both 16 bits. Will usually go into intermediate registers A and B.
  - Control signals:
    - RegWrite, determines whether data is being written to a register or not
    - RegRead, determines whether data is being read from a register or not
    - RegDst, determines which register data is being written to
    - RegData, determines the value that is written into the register specified by RegDst
    - PCWrite, to control writing to PC
    - SPWrite, to control writing to SP
    - SrcA, to control what goes into A
    - SrcB, to control what goes into B
- Memory
  - Input: Memory Address, 16 bits.
  - Output: Memory Data, 16 bits.
  - Control signals:
    - MemRead, determines whether data is being read from memory or not
    - MemWrite, determines whether data is being written to memory or not
    - MemSrc, determines where the address being used comes from
- ALU x1
  - Performs addition, subtraction, logical or, logical and, set-less-than, set-greater-than, and set-equal-to
  - Inputs: A and B (from intermediate registers A and B), each 16 bits
  - Control signals:

- ■ AluOp, to decide which operation the ALU will perform
    - ○ Output goes into AluOut (intermediate register), 16 bits
- Adder x2
    - ○ Adders used to add values to PC and SP
    - ○ Not controlled; they will always add to PC and SP, but the control signals PCWrite and SPWrite will determine which value is written to them
    - ○ Both inputs are 16 bits, output is 16 bits
- Control unit
    - ○ Sets all control signals based on instruction data
- Zero extender
    - ○ One 8-bit zero extender to extend the 8 bit immediate in the instruction data
    - ○ Input: 8 bits, output: 16 bits
- Sign extender
    - ○ One 8-bit sign extender to extend the 8 bit immediate in the instruction data
    - ○ Input: 8 bits, output: 16 bits
- Sign shifters
    - ○ A 2-bit left shifter for stack operations and a 4-bit left shifter for certain jump operations
    - ○ Input: 16 bits, output: 16 bits

Hardware Implementation

Register:

    Takes in data, a write control bit, and a clock. If the write control bit is 1 on the clock's rising edge, the register stores the data. If the write control bit is 0 or the clock is not on the rising edge, it does not store the data. If the write control bit is 0 on the clock's rising edge, it outputs its data to the output wire.

Memory:

    Take in data, a write control bit, an address, and a clock. If the write control bit is 1 on the clock's rising edge, the register stores the data into memory at the input address. If the write control bit is 0 on the clock's rising edge, it outputs the data at the input address into the output register MemVal. Each chunk of memory is 16 bits, and the total number of chunks is $2^{12} = 4096$.

ALU:

    Takes in two values, A and B, and an operation ("op") control. It performs the operation specified by the op control on the two inputs A and B, and puts the output into the output register AluOut. It also contains an overflow detector; if an addition or subtraction operation overflows, the overflow detector outputs 1.

Adder:

    Takes in two values, A and B, and outputs their sum to the output wire. It also contains an overflow detector; if the operation overflows, the overflow detector outputs 1.

Zero Extender:

    A zero extender takes in an 8 bit input and gives a 16 bit output, which has 8 leading zeroes concatenated with the original 8 bit input. If one creates it with a schematic, one only needs to connect the first 8 bits of the output to ground, and the remaining 8 to the 8 of the input. The logic is very simple in Verilog, and just needs to concatenate the input with zeroes.

Sign Extender:

    A sign extender works mostly the same as the zero extender, with an 8 bit input and 16 bit output, but logic determines whether there should be leading zeroes or leading ones concatenated with the original 8 bit input. If the most significant bit of the input is a 1, the leading 8 bits are 1's, while if it is a zero, the leading 8 bits are 0's.

Sign shifter:

    There are two left shift units; one is a left shift by two operation, and the other is a left shift by four. In the left

shift by two, the 16 bit input has the first two zeros removed, and that remaining 14 bits is concatenated with two trailing zeroes. For the left shift by four, a similar process occurs but with the first four bits removed, and four trailing zeroes concatenated onto the end.

Control Signals

MemWrite: 1-bit signal which determines whether or not data is
          written to or read from memory
 0: Read
 1: Write
MemDst: 3-bit signal which determines the address memory is accessed
        at
 000: PC
 001: Immediate address
 010: Address stored in Mary
 011: Address stored in Shelley
 100: Stack Pointer + 2
 101: Immediate left-shifted 2 + Stack Pointer
MemSrc: 2-bit signal which determines the data that is written to the
memory address specified by MemDst
 00: mary
 01: shelley
 10: ra
MaryWrite/ShelleyWrite/CompWrite/RAWrite/PCWrite/SPWrite/InstWrite: 1-
bit signals which determine whether or not data is written to the
register.
 0: Don't write
 1: Write
MarySrc: 2-bit signal which determines the value written into Mary.
 00: MemVal
 01: AluOut
 10: Shelley
 11: Immediate
ShelleySrc: 2-bit signal which determines the value written into
Shelley.
 00: MemVal
 01: Immediate
 10: Mary
 11: Nothing
RASrc: 1-bit signal which determines the value written into RA.
 0: MemVal
 1: PC+2
PCSrc: 3-bit signal which determines the value written into PC
 000: PC+2
 001: PC+immediate
 010: immediate (address)
 011: ra (return address)

```
      100: mary
      101: pc + left-shift4 mary
      110: imm if comp = 1, pc otherwise
      111: left-shift4 imm if comp = 1, pc otherwise
SPSrc: 2-bit signal which determines the value written into sp
      00: SP+0
      01: SP+2
      10: SP-2
      11: nothing
SrcA: 1-bit signal that determines what is written into A
      0: Mary
      1: SP
SrcB: 2-bit signal that determines what is written into B
      00: Shelley
      01: Zero-extended immediate
      10: Sign-extended immediate
      11: Sign-extended left-shifted immediate
AluOp: 3-bit signal that determines the operation performed by the ALU
on its inputs
      000: AND
      001: OR
      010: Add
      011: Sub
      100: SetLessThan
      101: SetGreaterThan
      110: SetEqualTo
      111: Output = input B (used for swap)
```

Datapath Diagram
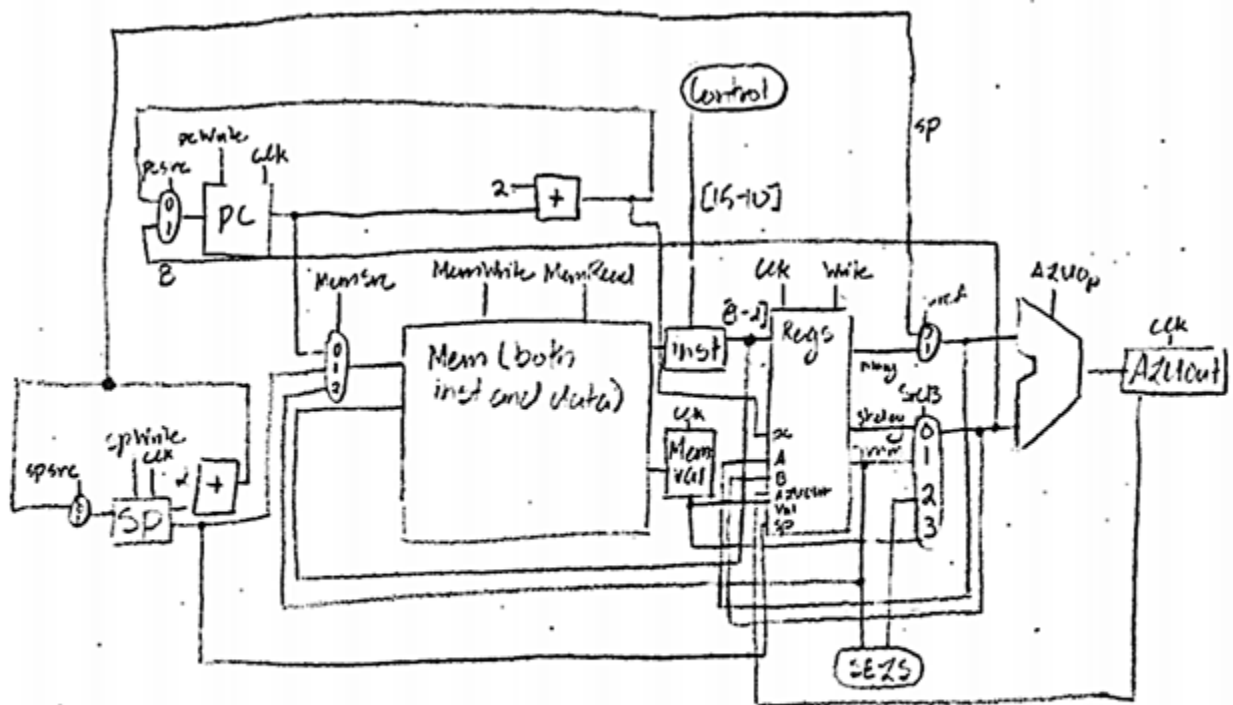
Note: wires are not drawn pointing to number on mux that corresponds to their control bits. Please see design docs for accurate control bits.
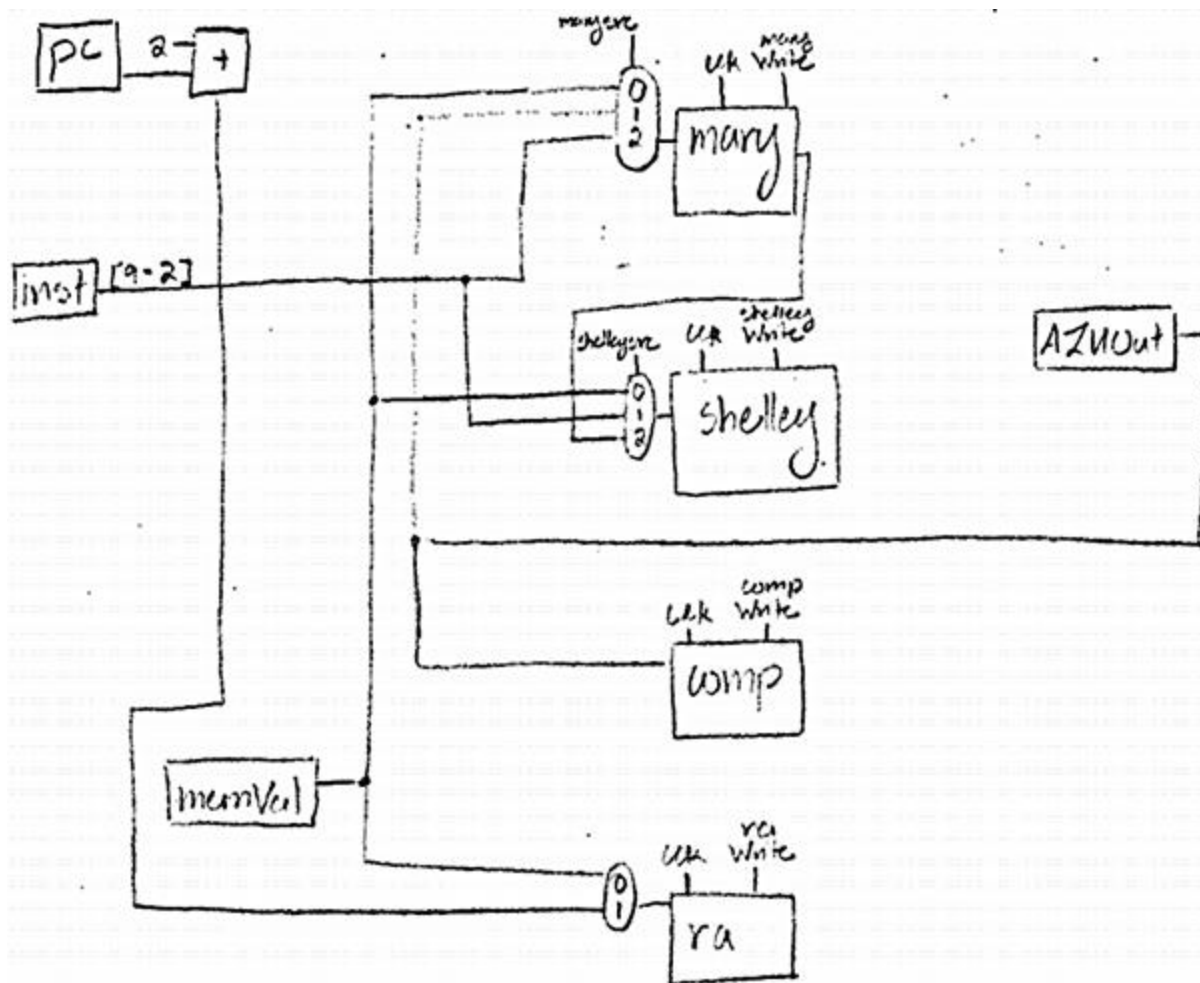
Integration Testing Overview

Each component takes input from a register and outputs to a register.
This is already built into the individual testing for each component.
To integrate them, the registers must simply be hooked up to both
their inputs and outputs (often with multiplexers to decide which ones
it makes use of).
We will slice Frankie into blocks: pc, sp, memory, ALU, our main regs,
and the muxes between the ALU and regs. We'll test each of these
individually, then together.

Integration Plan

1) We will test the PC block: PC, its adder, and its mux. We will check that PC has the right thing inside at the end of each test. We will then adapt the PC block for SP.
2) We will test main memory. We will test to make sure it has the correct values in inst and memVal.
3) We will add the pc and sp blocks to memory and test all three blocks together, checking the contents of inst at the tests' ends.
4) We will test the ALU. We will look at ALUOut to determine the success. We will test the Reg Block: mary, shelly, comp, and ra. We will not test them individually because they do not interact with each other enough to warrant it.
5) We will conduct full path testing to make sure the entire datapath is operating correctly for all instructions and all flag bits.

Control Unit

The control unit is constructed using a finite state machine design. Each combination of opcode, flag bit, and cycle produces a different control output, represented by a different state in the finite state machine, so all three must be kept track of and checked while the instruction is being executed. When the instruction reaches its last cycle, the control unit prepares the necessary control signals to load the next instruction.

All control signals are handled using this one control unit, and all of its functionality must be working as expected to be able to integrate all parts of the CPU. Currently, the control unit is almost fully constructed, with cases for all mnemonic and flagbit combinations, barring shift which needs to be discussed more within the group.

The control unit so far is in the implementation directory, within the ControlUnit subdirectory.

Control Unit Testing

The control unit will first be tested by manually setting all possible opcode/flag bit/cycle combinations in a Verilog test bench and verifying that the control bits are set correctly. Once it is confirmed that the basic logic is correct, it will be hooked up to

memory, and the instructions will be read from there instead. Once that is confirmed to be correct, the memory will be hooked up to PC; if it is still correct at this point it can be asserted with confidence that the control unit is fully operational.

Currently, the first stage of testing is complete. A testbench was created in Verilog that manually sets values for every possible combination of flagbit and OPCODE, and checks the output from the control unit. If the output matches the expected output for that particular OPCODE and flagbit, the test passes and a verification of that pass is displayed; otherwise, the mnemonic and a fail notification are displayed. All phase-1 tests of the control unit pass.

Details of the tests can be found in the contro_unit_test (accidentally misspelled during creation) found in the ControlUnit folder in the implementation directory for group 3V.

Control Unit Finite State Machine

For all states, control signals are assumed to be 0 unless asserted otherwise.
All instructions share the same first two states, which load the instruction from memory and process it in the control unit.

Cycle 1:
    PCWrite=1, PCSrc=000, MemWrite=0, MemDst=000, InstWrite=1
Cycle 2:
    All control signals 0, this cycle decodes the instruction and
    sets the new control bits

aput:
Cycle 3:
    MaryWrite=1, MarySrc=11

aput@:
Cycle 3:
    ShelleyWrite=1, ShelleySrc=01

sput:
Cycle 3:
    SPWrite=1, SPSrc=01, MemWrite=1, MemDst=100

aadd:
Cycle 3:
    SrcA=0, SrcB=01, AluOp=010
Cycle 4:
    MaryWrite=1, MarySrc=01

aadd@:
Cycle 3:
    SrcA=0, SrcB=00, AluOp=010
Cycle 4:
    MaryWrite=1, MarySrc=01

asub:
Cycle 3:
    SrcA=0, SrcB=01, AluOp=011

```
Cycle 4:
      MaryWrite=1, MarySrc=01
```

asub@:
```
Cycle 3:
      SrcA=0, SrcB=00, AluOp=011
Cycle 4:
      MaryWrite=1, MarySrc=01
```

spek:
```
Cycle 3:
      MemWrite=0, MemDst=101
Cycle 4:
      MaryWrite=1, MarySrc=00
```

spek@:
```
Cycle 3:
      MemWrite=0, MemDst=101
Cycle 4:
      ShelleyWrite=1, ShelleySrc=00
```

spop:
```
Cycle 3:
      MemWrite=0
      MemDst=100
Cycle 4:
      SPWrite=1, SPSrc=10, MaryWrite=1, MarySrc=00
```

rpop:
```
Cycle 3:
      MemWrite=0, MemDst=100
Cycle 4:
      SPWrite=1, SPSrc=10, RAWrite=1, RASrc=0
```

jimm:
```
Cycle 3:
      PCWrite=1, PCSrc=010
```

jimm@:
```
Cycle 3:
      PCWrite=1, PCSrc=001
```

jacc:

```
Cycle 3:
     PCWrite=1, PCSrc=100
```

jacc@:
```
Cycle 3:
     PCWrite=1, PCSrc=101
```

jcmp:
```
Cycle 3:
     PCWrite=1, PCSrc=110
```

jcmp@:
```
Cycle 3:
     PCWrite=1, PCSrc=111
```

jret:
```
Cycle 3:
     PCWrite=1, PCSrc=011
```

jfnc:
```
Cycle 3:
     PCWrite=1, PCSrc=010, RAWrite=1, RASrc=1
```

jfnc@:
```
Cycle 3:
     PCWrite=1, PCSrc=001, RAWrite=1, RASrc=1
```

cequ:
```
Cycle 3:
     SrcA=0, SrcB=01, AluOp=110
Cycle 4:
     CompWrite=1
```

cequ@:
```
Cycle 3:
     SrcA=0, SrcB=00, AluOp=110
Cycle 4:
     CompWrite=1
```
cles:
```
Cycle 3:
     SrcA=0, SrcB=01, AluOp=100
Cycle 4:
     CompWrite=1
```

cles@:

```
Cycle 3:
      SrcA=0, SrcB=00, AluOp=100
Cycle 4:
      CompWrite=1
```

cgre:
```
Cycle 3:
      SrcA=0, SrcB=01, AluOp=101
Cycle 4:
      CompWrite=1
```

cgre@:
```
Cycle 3:
      SrcA=0, SrcB=00, AluOp=101
Cycle 4:
      CompWrite=1
```

lorr:
```
Cycle 3:
      SrcA=0, SrcB=01, AluOp=001
Cycle 4:
      MaryWrite=1, MarySrc=01
```

lorr@:
```
Cycle 3:
      SrcA=0, SrcB=00, AluOp=001
Cycle 4:
      MaryWrite=1, MarySrc=01
```

land:
```
Cycle 3:
      SrcA=0, SrcB=01, AluOp=000
Cycle 4:
      MaryWrite=1, MarySrc=01
```

land@:
```
Cycle 3:
      SrcA=0, SrcB=00, AluOp=000
Cycle 4:
      MaryWrite=1, MarySrc=01
```

shfl:
```
      As of yet unspecified
```

shfl@:
    As of yet unspecified

shfr:
    As of yet unspecified

shfr@:
    As of yet unspecified

load:
Cycle 3:
    MemWrite=0, MemDst=001
Cycle 4:
    MaryWrite=1, MarySrc=00

load@:
Cycle 3:
    MemWrite=0, MemDst=011
Cycle 4:
    MaryWrite=1, MarySrc=00

stor:
Cycle 3:
    MemWrite=1, MemDst=001, MemSrc=00

stor@:
Cycle 3:
    MemWrite=1, MemDst=011, MemSrc=00

bkac:
Cycle 3:
    SPWrite=1, SPSrc=01, MemWrite=1, MemDst=100, MemSrc=00

bkac:
Cycle 3:
    SPWrite=1, SPSrc=01, MemWrite=1, MemDst=100, MemSrc=01

bkra:
Cycle 3:
    SPWrite=1, SPSrc=01, MemWrite=1, MemDst=100, MemSrc=10

swap:
Cycle 3:
    MaryWrite=1, MarySrc=10, ShelleyWrite=1, ShelleySrc=10