TEAM 3V

Maura Coriale

Ben Gothard

Matthew Lyons

Joy Stockwell

“Frankie”

CSSE232-02

Processor Design DocumentTable of Contents

[**Meet “Frankie”**](#_ulwlvfg1arus) **2**

[**Register file**](#_9xhbpi7l1i15) **3**

[**The “Clerval” Instruction Set Architecture**](#_1hd5r5bpmk3z) **4**

[**Writing an instruction**](#_8zb308lembrq) **5**

[**Converting an instruction to machine code**](#_z6rq939atpxp) **6**

[**Procedure calling conventions**](#_x8alebb4azph) **7**

[**Instructions Overview**](#_eyinzg1wpeds) **8**

[**Detailed Instructions Reference**](#_iasvc2cyapv3) **10**

[**I/O**](#_91lvcml9aksk) **22**

[**I/O is done using interrupts. Details will be added later.**](#_d91rz97pn1ry) **22**

[**Assembly Code Fragments**](#_8b8fnvy5hfwk) **23**

[**Euclid’s Algorithm**](#_6a1pr565f8ir) **24**

[**RTL**](#_yfrer4su50c) **27**

[**State Diagrams**](#_e0lwlrzc27mw) **35**

[**Shopping List**](#_fbv1n5qwntp6) **39**

[**Control Signals**](#_7ul1y3mlz7w6) **41**

# 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.

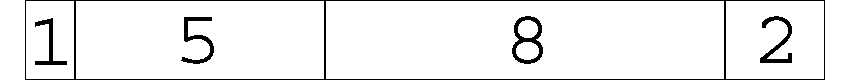
The op code for aadd is 00010.

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



# 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: 11000***

Empty instruction that does nothing and is always skipped.

The flag bit has no effect on noop.

# 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

# 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 |  |
| 0x0002 | spop | get n into the main accumulator | 10010100000000 |
| 0x0004 | cequ 1 | put 1 in the comp reg if n is equal to 1 | 00101100000001 |
| 0x0006 | swap | get ready for the loop or return by putting m into the main accumulator | 01011000000000 |
| 0x0008 | jcmp ENDgcd | if 1 is in the comp reg (n == 1) then jump to the end where we return m | 00100100011100 |
| 0x000A | LOOPgcd:  cequ 0 | puts 1 into the comp reg if m is 0 | 00101100000000 |
| 0x000C | jcmp ENDswap | if m is 0, the comp reg holds 1, so jump out of the loop and return a = n | 00100100011010 |
| 0x000E | 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 | 00110000000001 |
| 0x0010 | jcmp ELSEgcd | jump to the else clause if !(a > b) --> !(n > m) | 00100100011000 |
| 0x0012 | swap | if (a > b) --> (n > m), puts n into main accumulator so we can change its val | 01011000000000 |
| 0x0014 | 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 to be the return value |  |
| 0x0016 | jimm ENDgcd | jump to the end |  |
| 0x0018 | ELSEgcd:  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 |  |
| 0x001A | ENDswap:  swap | n was in the backup; now it is in the main to be returned |  |
| 0x001C | ENDgcd:  jret | jump to the addr in ra (ie a line in relPrime) |  |
|  |  |  |  |
| 0x001E | main: |  |  |
| 0x0020 | bkra | Back up the ra on the stack before putting the args on the stack so that GCD can get back to relPrime |  |
| 0x0022 | sput nVal | Put n on the stack |  |
| 0x0024 | aput 2 | Put 2 (i.e. mVal) in the accumulator |  |
| 0x0026 | swap | put 2 in the backup accumulator |  |
| 0x0028 | LOOPrp: jfnc 0 GCD | go to the function in the other file; automatically sets ra to the address of this line |  |
| 0x002A | 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. |  |
| 0x002C | jcmp ENDrp | continue in the loop unless GCD returned 1 which is to say, break out of the loop if GCD returned 1 |  |
| 0x002E | swap | if GCD didn't return 1, get out of the backup accumulator and into the main accumulator |  |
| 0x0030 | aadd 1 | add 1 to m |  |
| 0x0032 | swap | put the new m = mOld + 1 back in the backup accumulator so it is the arg for the next time we call GCD |  |
| 0x0034 | jimm LOOPrp | do the loop again |  |
| 0x0036 | ENDrp:  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 |  |
| 0x0038 | rpop | restore ra to the address of whatever called relPrime |  |
| 0x003A | jret | jump to wherever called relPrime |  |

# 

# 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] | | | | | |
| A = mary  B = shelley/imm | A = mary,  B = shelley/  imm | A = sp  B = imm | B = LS(SE(imm))  or B = imm | A = mary  B = shelley | val = Mem[imm]  (only load) |
| ALUOUT = A OP B | ALUOUT = A OP B | ALUOUT = A OP B | PC = B |  |  |
| mary = ALUOUT | cmp = ALUOUT | mary/shelley = ALUOUT |  | mary = B  shelley = A | mary = val 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]

sp = sp + 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]

B = imm\*2

B = B + sp

val = Mem[sp]

mary = val

spop:

pc = pc + 2

inst = Mem[PC]

flagbit = inst[15]

OPCODE = inst[14, 10]

imm = inst[9, 2]

val = Mem[sp]

sp = sp + 2

mary = val

rpop:

addr = mem[sp]

sp -= 1 word

reg[ra] = addr

jimm:

PC = imm

jimm@:

target = imm << 4

PC = target

jacc:

PC = mary

jacc@:

target = mary << 4

PC = target

jcmp:

if reg[cmp] == 1:

PC = imm

jcmp@:

if reg[cmp] == 1:

target = imm << 4

PC = target

jret:

PC = reg[ra]

jfnc:

reg[ra] = PC + 1 word

PC = imm

jfnc@:

reg[ra] = PC + 1 word

target = imm << 4

PC = target

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

load:

pc = pc + 2

inst = Mem[pc]

flagbit = inst[15]

OPCODE = inst[14, 10]

val = Mem[imm]

mary = val

load@:

pc = pc + 2

inst = Mem[pc]

flagbit = inst[15]

OPCODE = inst[14, 10]

val = Mem[shelly]

mary = val

stor:

Same problem as with load

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]

A = mary

B = shelly

mary = B

shelly = A

# 

# 

# States for Each Instruction Type

|  |  |
| --- | --- |
| Stack | |
| pc = pc + 2 | PCWrite = 0 |
| inst = Mem[pc] | MemRead = 1 |
| flagbit = inst[15]  OPCODE = inst[14, 10]  imm = inst[9, 2]  A = sp, B = SE(LS(imm)) | RegRead = 1  srcA = sp  srcB = SE(LS(imm)) |
| ALUOUT = A + B | ALUOP = add |
| memVal = Mem[ALUOUT)  sp = sp + 2 (only for spop)  mary = memVal | regWrite = 1  memToReg = 1  regDest = Mary  spWrite = 1 |

|  |  |
| --- | --- |
| Compare | |
| pc = pc + 2 | PCWrite = 0 |
| inst = Mem[PC] | memRead = 1 |
| flagbit = inst[15]  OPCODE = inst[14, 10]  imm = inst[9, 2]  A = Mary  B = shelley (1) OR B = imm | regRead = 1  srcA = mary  srcB = shelley OR srcB = imm |
| ALUOUT = A - B | ALUOP = sub |
| cmp = ALUOUT | regWrite = 1  regDest = cmp |

|  |  |
| --- | --- |
| Jump (to function) | |
| 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  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] |  |
| A = mary  B = shelley OR B = imm | srcA = mary  srcB = shelley OR srcB = imm |
| mary = B | regWrite = 1  regDest = mary  regData = B |
| shelley = A | 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

* Register file
  + Contains Mary, Shelley, RA, and Comp
  + 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
* 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
* Single registers x7
  + Intermediate registers to hold data in multicycle. Includes PC, SP, A, B, AluOut, Inst, and MemVal.
  + Control signals:
    - 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
* 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 (which are separate from the main RegFile)
  + 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

# 

# Control Signals

PCWrite: 2-bit signal which determines the value written into PC

00: PC+2

01: PC+immediate

10: immediate (address)

11: ra (return address)

SPWrite: 2-bit signal which determines the value written into sp

00: SP+0

01: SP+2

10: SP-2

11: nothing

MemRead: 1-bit signal which determines whether or not data is read

from memory

0: Don’t read

1: Read

MemWrite: 1-bit signal which determines whether or not data is

written to memory

0: Don’t write

1: Write

MemSrc: 2-bit signal which determines the address memory is accessed

at

00: PC

01: Immediate address

10: Address stored in Mary

11: Address stored in Shelley

RegRead: 1-bit signal which determines whether or not data is read

from the regfile

0: Don’t read

1: Read

RegWrite: 1-bit signal which determines whether or not data is

written to the regfile

0: Don’t write

1: Write

RegDst: 2-bit signal that determines which register data will be

written to

00: Mary

01: Shelley

10: RA

11: Comp

RegData: 2-bit signal that determines where the data being written to

the regfile is coming from

00: AluOut

01: MemVal

10: A

11: B

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