CSE 141L Milestone 3

John Adams A16499049

# Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

* Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
* Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

John P Adams

# 0. Team

John Adams.

# Introduction

Name: TMR (too many registers)

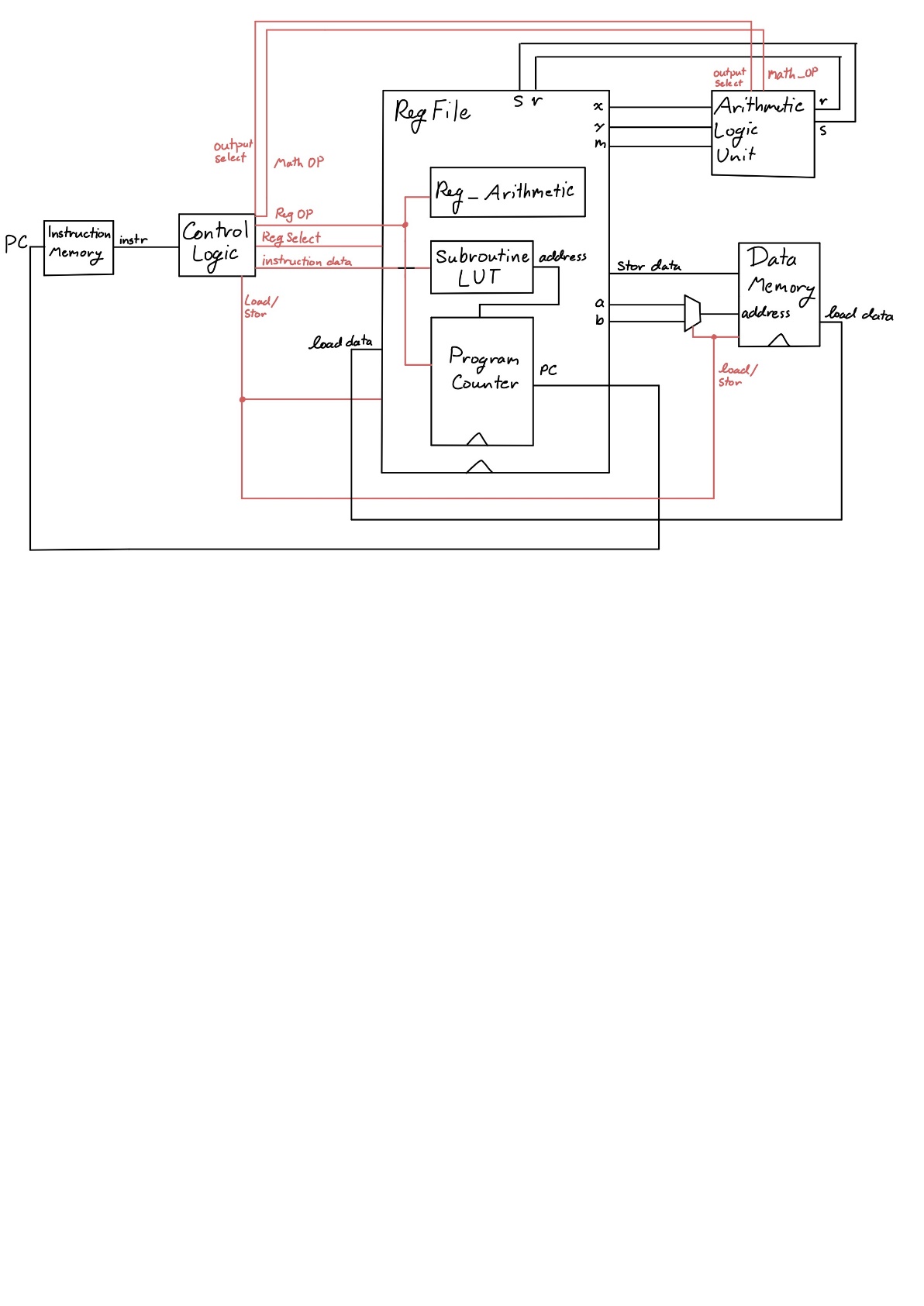
Philosophy: Use specialized registers so you can do more “stuff” without having to specify where it comes from.

Goals: make a cpu that was:

* easy to code for (lots of registers, designed with for loops in mind, 8-bit literals, many math operations, built in subroutines)
* entirely from scratch, because why would you learn how to use an api when you can spend twice as long writing and debugging your own version.

My cpu is a Load-Store (register-register) architecture. Although by using one of the address registers, it would be possible to implement a stack in software with only 2 instructions for push (stor b {reg}, incr b) and pop (load b {reg}, decr b).

# Architectural Overview



# Machine Specification

## Instruction formats

|  |  |  |
| --- | --- | --- |
| **TYPE** | **FORMAT** | **CORRESPONDING INSTRUCTIONS** |
| I | 5-bit OP code, 4-bit val | vall, valh, jtsr |
| R | 5-bit OP code, 4-bit reg | movc, movd,  movm, movn, movx, movy,  mova, movb, movi, movj,  movk, movv, movz,  bizr, bnzr, incr, decr, flip |
| F | 5-bit OP code, 4-bit operand | mthr, mths, func |
| I’ | 5-bit OP code, 1-bit reg, 3-bit val | jizr, jnzr, lslc, lsrc, |
| R’ | 5-bit OP code, 1-bit reg, 3-bit reg | load, stor |

There is one unused instruction (5-bit opcode that is unused) allowing for one additional instruction to be added in the future.

There are 8 unused function operands.

There is one unused math operation.

## Operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **TYPE** | **BIT BREAKDOWN** | **EXAMPLE** | **NOTES** |
| vall  value register low nibble | I | 5-bit OP code (00000)  4-bit value (XXXX) | # let v hold b00110000  vall 4’b1010  # v now holds b00111010 | takes place of the movr instruction |
| valh  value register high nibble | I | 5-bit OP code (00001)  4-bit value (XXXX) | # let v hold b00110000  valh 4’b1010  # v now holds b10100000 | takes place of the movs instruction |
| jtsr  jump to subroutine | I | 5-bit OP code (11000)  4-bit value (XXXX) | # let LUT[3]= 10’b0100010111  # let pc = 10’b0000001001  # let l = 10’b0000000000  jtsr 4’b0011  # pc = 10’b0100010111  # l = 10’b0000001010 | stores pc+1 in link (l) and sets pc to LUT[XXXX] |
| mov$  move to register | R | 1-bit OP code (0)  4-bit register (XXXX)  4-bit register (XXXX) | # let v hold b10001000  # let x hold b00000000  movx v  # x now holds b10001000 | mov is considered a single instruction, but is encoded at mov$ where $ is the name of the destination register (excluding r/s)  **NOTE:** moving a register to itself zeros out the register |
| bizr  branch if zero | R | 5-bit OP code (10110)  4-bit register (XXXX) | # let r hold b00000000  # let pc hold 10’b1000100100  # let z hold b11000000  bizr r  # pc now at 10’b1011000000 | bizr/bnzr can only update the lower 8-bits of the program counter. |
| bnzr  branch if not zero | R | 5-bit OP code (10111)  4-bit register (XXXX) | # let r hold b00000000  # let pc hold 10’b1000100100  # let z hold b11000000  bnzr r  # pc now at 10’b1000100101  # branch was not taken | bizr/bnzr can only update the lower 8-bits of the program counter. |
| incr  increment register | R | 5-bit OP code (10010)  4-bit register (XXXX) | # let a hold b01010000  incr a  # a now holds b01010001 | incrementing r, s registers has no effect |
| decr  decrement register | R | 5-bit OP code (10011)  4-bit register (XXXX) | # let b hold b10000000  decr b  # b now holds b01111111 | decrementing r, s registers has no effect |
| flip | R | 5-bit OP code (11110)  4-bit register (XXXX) | # let n hold b00000000  # let c hold b00001011  flip c  # n now holds b00000100 | if reg[4] == 1: destination is n  otherwise: destination is m  xor the reg[2:0]th bit in the destination |
| amp  logical and | F | 5-bit OP code (1101X)  4-bit operation (0000) | # let x hold b11111100  # let y hold b00111111  mthr amp  # r now holds b00111100 |  |
| lor  logical or | F | 5-bit OP code (1101X)  4-bit operation (0001) | # let x hold b11111100  # let y hold b00111111  mthr lor  # r now holds b11111111 |  |
| flp  logical not | F | 5-bit OP code (1101X)  4-bit operation (0010) | # let x hold b11111100  mthr flp  # r now holds b00000011 |  |
| eor  logical exclusive or | F | 5-bit OP code (1101X)  4-bit operation (0011) | # let x hold b11111100  # let y hold b00111111  mthr eor  # r now holds b11000011 |  |
| rsc  right shift carry | F | 5-bit OP code (1101X)  4-bit operation (0100) | # let x hold b10111100  # let y hold b00111111  mthr rsc  # r now holds b11011110 | res = {y[0],x[7:1]} |
| lsc  left shift carry | F | 5-bit OP code (1101X)  4-bit operation (0101) | # let y hold b00111111  mthr lsc  # r now holds b00011111 | res = {x[6:0,]y[7]} |
| ror  rotate right | F | 5-bit OP code (1101X)  4-bit operation (0110) | # let x hold b11101000  # let y hold b00000011  mthr rol  # r now holds b01000111 | rotate x by value in y[2:0] |
| add  algebraic add | F | 5-bit OP code (1101X)  4-bit operation (0111) | # let x hold b00001111  # let y hold b00000001  mthr add  # r now holds b00010000 |  |
| sub  algebraic subtract | F | 5-bit OP code (1101X)  4-bit operation (1000) | # let x hold b00010110  # let y hold b00000111  mthr sub  # r now holds b00001111 |  |
| eql8  check for byte equality | F | 5-bit OP code (1101X)  4-bit operation (1001) | # let x hold b10101000  # let y hold b10101111  mthr eql8  # r now holds b00000000 |  |
| eql5  check for upper 5-bit equality | F | 5-bit OP code (1101X)  4-bit operation (1010) | # let x hold b10101000  # let m hold b10101111  mthr eql5  # r now holds b00000001 | tests for x[7:3] = m[7:3]  **note** this is the only math instruction that uses a register other than only x, y |
| revx  reverse byte x | F | 5-bit OP code (1101X)  4-bit operation (1011) | # let x hold b11110000  mthr revx  # r now holds b00001111 |  |
| revy  reverse byte y | F | 5-bit OP code (1101X)  4-bit operation (1100) | # let y hold b10100011  mthr revy  # r now holds b11000101 |  |
| parx  compute x parity | F | 5-bit OP code (1101X)  4-bit operation (1101) | # let x hold b00000111  mthr parx  # r now holds b00000001 | computes the xor of all bits in x |
| pary  compute y parity | F | 5-bit OP code (1101X)  4-bit operation (1110) | # let y hold b10101010  mthr pary  # r now holds b00000000 | computes the xor of all bits in y |
| ljp$  long jump | F | 7-bit op code (1111100)  2-bit operation (XX) | # let z hold 11111111  func ljp3  # pc now at 10’b1111111111 | ljp is part of the function group, and can be called by specifying func ljp$ where $ is the decimal representation of the upper 2 bits of the next program counter value. |
| strl  start low | F | 9-bit op code  (111111100) | # start\_add = 10’b0000000000  # z = 11111010  func strl  # start\_add = 10’b1000000000 | sets the upper two bits of the start\_address using z[1:0] |
| strh  start high | F | 9-bit op code  (111111101) | # start\_add = 10’b0000000000  # z = 11111010  func strh  # start\_add = 10’b0011111010 | sets the lower 8 bits of the start\_address using z |
| rfsr  return from subroutine | F | 9-bit op code  (111111110) | # let pc = 10’b1010000001  # let l = 10’b0000010000  func rfsr  # pc = 10’b0000010000 | loads link register into program counter  can be used for branch always |
| done  set done flag | F | 9-bit op code  (111111111) | func done | sets done flag outside of processor |
| lslc  logical shift left with carry | I’ | 5-bit OP code (11100)  1-bit register (X)  3-bit value (XXX) | # let m = b00111100  # let n = b10000000  lslc sm 1  # m now holds b01111001 | shifts (m:0/n:1) left by val, shifts in val highest bits from the other register (n/m) |
| lsrc  logical shift right with carry | I’ | 5-bit OP code (11101)  1-bit register (X)  3-bit value (XXX) | # let m = b00000101  # let n = b00001000  lsrc sn 3  # n now holds b10100001 | shifts (m:0/n:1) right by val, shifts in val lowest bits from the other register (n/m) |
| jizr  jump if zero | I’ | 5-bit OP code (11101)  1-bit register (X)  3-bit value (XXX) | # let r = b00000000  # let pc = 10’b0010000000  jizr r 2  # pc = 10’b0010000100  # jump taken | jump pc counter forward by 3’b \* 2  only works with r:0, s:1 registers  a value of 3’b000 results in a 4’b1000 jump |
| jnzr  jump if not zero | I’ | 5-bit OP code (11101)  1-bit register (X)  3-bit value (XXX) | # let s = b00000000  # let pc = 10’b0010000000  jnzr s 2  # pc = 10’b0010000001  # jump not taken | jump pc counter forward by 3’b \* 2  only works with r:0, s:1 registers  a value of 3’b000 results in a 4’b1000 jump |
| load  load register | R’ | 5-bit OP code (11101)  1-bit register (X)  3-bit register (XXX) | # let c = b11110000  # let mem[0] = b10101010  # let b = b00000000  load b c  # c = b10101010 | can select between a, b addresses  load into only c, d, m, n, x, y |
| stor  store register | R’ | 5-bit OP code (11101)  1-bit register (X)  3-bit register (XXX) | # let r = b00111100  # let mem[4] = b10101010  # let a = b00000100  stor b r  # mem[4] = b00111100 | can select between a, b addresses  stor from only r, s, c, d, m, n, x, y |

## Internal Operands

There are 16 registers (since I needed 11, I added the other 5 to use up the rest of the 4-bits needed to pick between more than 8)

Several registers are special-purpose. **NOTE:** at power-on, registers are initialized to their encoding value (4’h0 – 4’hf)

address: a, b - specify the memory address for load and store instructions

math: x, y - primary inputs for ALU

result: r, s - read-only result registers from ALU

bitwise: m, n - registers for bit-wise operations, and additional ALU input parameters (for special operations)

value: v - location for literal value instructions

branch: z - branch target

link: l - holds previous pc location after a jump to subroutine instruction

generic: c, d, i, j, k - generic registers for counters and other things.

Register encoding values:

3-bit accessible: 0: r 1: s 2: c 3: d 4: m 5: n 6: x 7: y <= these can be using in load/store instructions

4-bit accessible: 8: a 9: b a: i b: j c: k d: v e: z f: l

## Control Flow (branches)

Note: cycles are a measure of jumping to a **user-specified address** (i.e. directly from a literal value).

There are two conditional branches (branch if zero and branch if not zero) which can update the lower 8-bits of the program counter to the value in the branch (z) register based on the value of any register (range:256, precision:1, cycles: 4).

There are 4 ways to update the program counter in code

* jmp instructions add (3’b \* 2) from the program counter (range: 16, precision: 2, cycles: 1)
* jtsr (jump to subroutine) set the pc register to any of 16 predefined addresses (range:1024, precision:1, cycles:1)
  + also stores the previous pc address + 1 in the link (l) register
  + using multiple jtsr instructions in sequence does not store multiple values in the link register.
* rfsr (return from subroutine) restore the value of the pc register to the value in the link (l) register (range:1024, precision:1, cycles:1)
* long-jump functions modify the upper 2-bits of pc and load the branch (z) register into lower 8-bits (range:1024, precision:1, cycles:4)
  + they function as branch-always instructions

## Addressing Modes

Memory is handled indirectly. Memory addresses must be stored in either of the two 8-bit address registers (a, b).

Load instructions can read either address register and store into only one of the 3-bit accessible registers (c, d, m, n, x, y) excluding the read-only result registers (r, s).

Store instructions can also read either address register and then store from only one of the 3-bit accessible registers (r, s, c, d, m, n, x, y).

# Programmer's Model [Lite]

**4.1** There are many registers, each of which supports increment/decrement and can be the cmp source for branch instructions. This allows for simultaneous counters to exist at the same time, without sacrificing too much space for other important values. This is especially true of the memory address registers, allowing for a 2-instruction increment-load or decrement-load style sequential memory access.

The math x/y registers and result r/s registers are best suited to a particular workflow, that being load x, load y, compute > r, mov x <- r, compute > s, etc. A good example of this is a double-precision (16-bit) xor, which can be accomplished in 7 instructions.

The relatively small distance provided by the conditional-relative-jump instructions makes it easier to do conditional branching forward, with one larger absolute jump back to the beginning of the program for a looping, repetitive processes.

The inclusion of a link register allows for simple 1-instruction branch to subroutines, and a 1-instruction return from that subroutine back to the main thread of execution. This encourages modularity of code, such that up to 16 different smaller programs can be executed from the main program. Additionally, only their starting positions have to be noted, as the return-from-subroutine instruction utilizes the link register to continue execution.

**4.2** The arm instruction set is proprietary protected by copyright and patent such that a license is required to modify and reproduce the same instruction set. I got around this by not looking too much at the arm instruction set. I came up with my own set of instructions needed for the programs, added some more unique instructions, and made my ISA take advantage of special use registers, which are not part of the arm ISA.

**4.3** No, the ALU is not used in non-arithmetic instructions. There are two additional simplified arithmetic logic units inside the register file that handle calculating relative jumps and register increment/decrement operations. This reduces the amount of re-routing required to update registers (and since my ALU uses fixed input and output registers it is easier to implement it as a single-purpose ALU)

# Individual Component Specification

## **Top Level**

Module file name: top\_level.sv

### **Functionality Description**

Consists of wires and the instantiations of the processor components. It also includes one mux for selecting between address registers (a/b).

### **Schematic**

## **Program Counter**

Module file name: program\_counter.sv

Module testbench file name: tb\_program\_counter.sv

### **Functionality Description**

The program counter is a separate module located inside the register file. The register file decodes instructions from the controller and passes in a series of flags to the program counter that determine how it updates the counter.

Note: it is inside the register file module, but is entirely self-contained, I am just too lazy to add another 8 outputs to the register file to put the program counter on the top level, it’s a very error prone process.

### **(Optional) Testbench Description**

Test bench sets all the flags individually with different address inputs to test for branch conditions and standard incrementation of the counter.

### **Schematic**

### A computer screen shot of a diagram Description automatically generated

### **(Optional) Timing Diagram**

A screen shot of a computer

Description automatically generated

## **Instruction Memory**

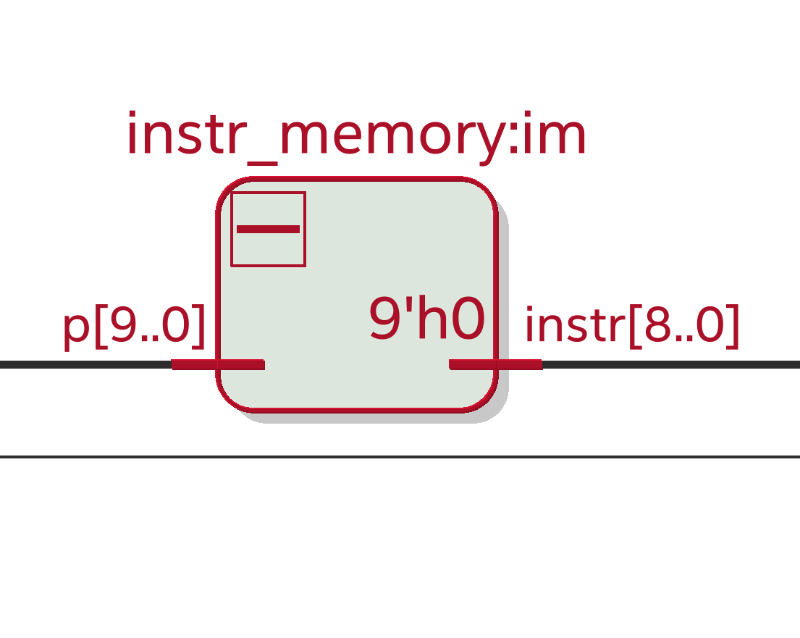
Module file name: instr\_memory.sv

### **Functionality Description**

Stores the instructions used in the programs. It supports a size up to 1024 bytes, accessible using a 10-bit program counter.

It is a read only memory, and does not support writing during execution.

### **Schematic**



## **Control Decoder**

Module file name: control\_logic.sv

### **Functionality Description**

This module decodes a 9-bit instruction into a

* Math operation selection and ALU flags
* Register File operation selection and flags (branch conditions)
* Memory Load/Store and address selection bit
* Done flag

### **Schematic**

**A diagram of a computer scheme

Description automatically generated**

## **Register File**

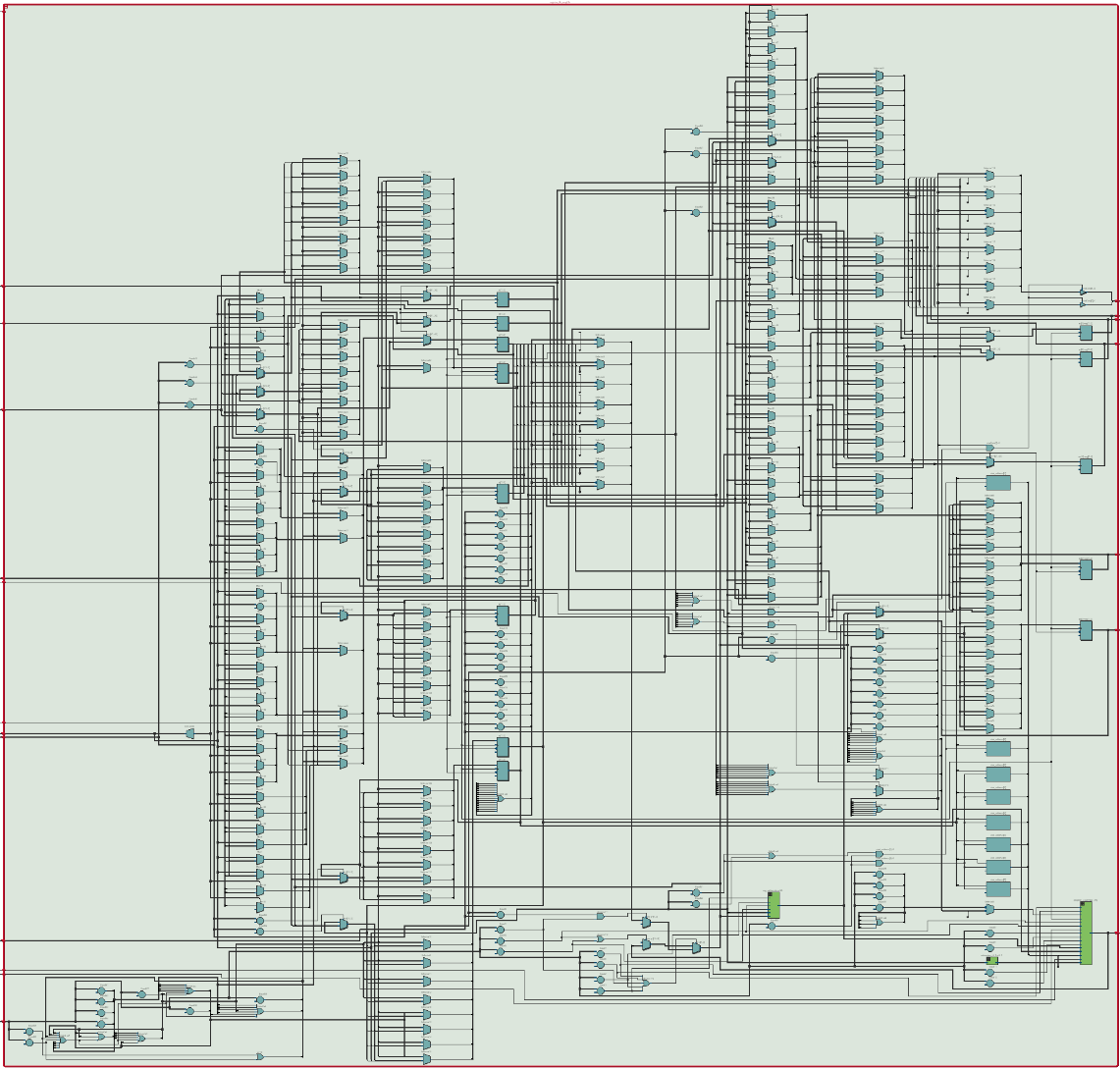
Module file name: register\_file\_r.sv

### **Functionality Description**

Holds all the registers, and has several sub-modules: program counter, subroutine look up table, register arithmetic.

The register file decodes register instructions from the control logic module (move, increment/decrement, branch/jump, subroutine, load/store, bitwise updates, and in-place shifting) and sets various flags according to the instruction to be executed. There are 3 distinct sections, one for setting flags, one for preparing data for the instruction, and one for executing the instruction (updating registers) on the following clock pulse.

### **Schematic**



## **ALU (Arithmetic Logic Unit)**

Module file name: arithmetic\_logic.sv

Module testbench file name: tb\_arithmetic\_logic.sv

### **Functionality Description**

Does one of 16 different math and logic operations on the two input registers x, y (except for 1 instruction that uses x and m). The result of the operation is stored in either the r or s output register dependent on the selection bit.

### **(Optional) Testbench Description**

The testbench performs each of the operations with 2 different pairs of inputs of x and y (except for test for upper-5-bit equality, which uses x and m).

### **ALU Operations**

logical and, logical or, logical not (x), logical exclusive or, logical right shift with carry-in (y>>x), logical left shift with carry-in (x<<y), rotate left (x), rotate right (x), add, subtract (x-y), test for equality (x==y), test for upper-5-bit equality (x[7:4] = m[7:4]), reverse x, reverse y, parity of x, parity of y.

### **Schematic**

A blueprint of a machine

Description automatically generated

### **(Optional) Timing Diagram**

A screenshot of a computer

Description automatically generated

## **Data Memory**

Module file name: data\_memory.sv

### **Functionality Description**

Takes an address in, load/store flag, and data\_in/data\_out and either writes data into the specified address or loads data out of the address.

### **Schematic**

A computer diagram with text and a purple square

Description automatically generated with medium confidence

## **Look Up Tables**

Module file name: subroutine\_LUT.sv

### **Functionality Description**

A 4-bit lookup table which outputs 10-bit addresses to the program counter during jump to subroutine instructions.

### **Schematic**

A screenshot of a computer

Description automatically generated

## **Other Modules (if necessary)**

Module file name: reg\_arithmetic.sv

### **Functionality Description**

Takes in a 10-bit value and does one of 3 arithmetic operations: increment, decrement, add 4’bXXX0 (specified by the instruction). This enables any register to be incremented, as well as handles local jumps of the program counter. Contains Full-Adder modules (which are self explanatory).

### **Schematic**

A computer screen shot of a diagram

Description automatically generated

# Program Implementation

Note: these look perfectly fine in a text editor, word is causing some ridiculous spacing. (just make sure tab-width=4)

## Program 1 Assembly Code

/\*init\*/ vall, 4'b1110

valh, 4'b0001

mova, v // a= 30 (decr > load)

vall, 4'b1011

valh, 4'b0011

movb, v // b= 59 (stor > decr)

vall, 4'b1001

valh, 4'b0000

movz, v // z= 8'b00001001 = 9 (branch target)

/\*load\*/ decr, a

load, sa, d[2:0] // {d,c}= data

decr, a

load, sa, c[2:0]

/\*p8\*/ movm, c

movn, d // {c,d} => {n,m} bitwise

lslc, sn, 3'd4 // n= {0,b11,b10,b9,b8,b7,b6,b5}

movm, m // m= 8'b0

lslc, sn, 3'd1 // n= {b11,b10,b9,b8,b7,b6,b5,0}

movx, n

mthr, parx // r= {0000\_000, p8}

movy, r

mths, lor // s= {b11,b10,b9,b8,b7,b6,b5,p8}

movd, s // s => d

/\*plce\*/ movm, c

movn, n

lslc, sm, 3'd4 // m= {b4,b3,b2,b1,0000}

vall, 4'b0000

valh, 4'b0001 // v= 8'b0001\_0000

movx, v

movy, m // {0001\_0000} & {b4,b3,b2,b1,0000} => r

mthr, amp // r= {000,b1,0000}

valh, 4'b1110 // v= 1110\_0000

movx, v // {1110\_0000} & {b4,b3,b2,b1,0000} => s

mths, amp // s= {b4, b3, b2, 0\_0000}

vall, 4'b0001

valh, 4'b0000

movx, r

movy, v

mthr, ror // r= {0000,b1,000}

movx, r

movy, s // {0000,b1,000} & {b4, b3, b2, 0\_0000}

mthr, lor // r= {b4,b3,b2,0,b1,000}

movc, r // r => c

/\*p4\*/ vall, 4'b0000

valh, 4'b1111 // mask = 11110000

jtsr, 4'd0 // subroutine 0 r= {0000\_000,p4}

movx, r

vall, 4'b0100

valh, 4'b0000

movy, v // y= 0000\_0100

mthr, ror // r= {000,p4,0000}

movx, r

movy, c // {000,p4,0000} | {b4, b3, b2, 0, b1, 000}

mthr, lor // r= {b4,b3,b2,p4,b1,000}

movc, r

/\*p2\*/ vall, 4'b1100

valh, 4'b1100 // mask = 11001100

jtsr, 4'd0 // subroutine 0 r= {0000\_000,p2}

movx, r

vall, 4'b0110

valh, 4'b0000

movy, v // y= 0000\_0110

mthr, ror // r= {0000\_0,p2,00}

movx, r

movy, c // {0000\_0,p2,00} | {b4,b3,b2,p4,b1,000}

mthr, lor // r= {b4,b3,b2,p4,b1,p2,00}

movc, r

/\*p1\*/ vall, 4'b1010

valh, 4'b1010 // mask = 10101010

jtsr, 4'd0 // subroutine 0 -> r= {0000\_000,p1}

movx, r

vall, 4'b0111

valh, 4'b0000

movy, v // y= 0000\_0111

mthr, ror // r= {0000\_00,p1,0}

movx, r

movy, c // {0000\_00,p1,0} | {b4,b3,b2,p4,b1,p2,00}

mthr, lor // r= {b4,b3,b2,p4,b1,p2,p1,0}

movc, r

/\*p0\*/ vall, 4'b1111

valh, 4'b1111 // mask = 11111111

jtsr, 4'd0 // subroutine 0 -> r= {0000\_000,p0}

movx, r

movy, c // {0000\_000,p0} | {b4,b3,b2,p4,b1,p2,p1,0}

mthr, lor // r= {b4,b3,b2,p4,b1,p2,p1,p0}

movc, r

/\*stor\*/ stor, sb, d[2:0]

decr, b

stor, sb, c[2:0]

decr, b

/\*comp\*/ bnzr, a // if a!=0, go to line 9.

func, done

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

/\*sub0\*/ movx, v // subroutine calc parity using mask in v, store in r

movy, c

mthr, amp // mask c

movy, d

mths, amp // mask d

movx, r

movy, s

mthr, parx // calculate byte parity

mths, pary

movx, r

movy, s

mthr, eor // calculate total parity

func, rfsr

## Program 2 Assembly Code

/\*init\*/ vall, 4'b1110

valh, 4'b0001

mova, v // a= 30 (decr > load)

vall, 4'b1100

valh, 4'b0011

movb, v // b= 60 (decr > stor)

vall, 4'b1001

valh, 4'b0000

movz, v // z= 8'b00001001 = 9 (branch target)

decr, b

load, sb, d[2:0]

decr, b

load, sb, c[2:0] // {d,c}= data

movm, m // m= 00000000 (will hold parity bits)

movn, n // n= 00000000

/\*p8\*/ movx, d

mthr, parx

movn, r

lsrc, sm, 3'b001 // m= {p8,000\_0000}

/\*p4\*/ vall, 4'b0000

valh, 4'b1111 // mask = 11110000

jtsr, 4'd0 // r= {0000\_000,p4}

movn, r

lsrc, sm, 3'b001 // m= {p4,p8,00\_0000}

/\*p2\*/ vall, 4'b1100

valh, 4'b1100 // mask = 11001100

jtsr, 4'd0 // r= {0000\_000,p2}

movn, r

lsrc, sm, 3'b001 // m= {p2,p4,p8,0\_0000}

/\*p1\*/ vall, 4'b1010

valh, 4'b1010 // mask = 10101010

jtsr, 4'd0 // r= {0000\_000,p1}

movn, r

lsrc, sm, 3'b001 // m= {p1,p2,p4,p8,0000}

/\*p0\*/ vall, 4'b1111

valh, 4'b1111 // mask = 11111111

jtsr, 4'd0 // r= {0000\_000,p0}

/\*orgz\*/ movx, m

mths, revx // s= {0000,p8,p4,p2,p1}

movm, c

movn, d // {n,m}= data

movx, x // x= 8'b00000000 (jump condition)

vall, 4'b0000 // v= 8'b????0000

/\*ifp0\*/ jizr, sr, 3'b011

valh, 4'b0100 // if p=1, 1 error

flip, s // correct {n,m} by r[3:0]

mthr, revx

jizr, sr, 3'b100

func, noop

/\*elif\*/ jizr, ss, 3'b010

valh, 4'b1100 // if p=0 && (p1:8!=0) 2 errors

mthr, revx

jizr, sr, 3'b001

/\*othr\*/ valh, 4'b0000 // if p=0 && p1:8=0

movk, v

movk, v // k= {F1,F0,00\_0000}

movc, m

movd, n // {d,c} = corrected data

/\*deco\*/ vall, 4'b1000

valh, 4'b1110

movx, v

movy, c // {1110\_1000} & {b4, b3, b2, p4, b1, p2, p1, p0}

mthr, amp // r= {b4,b3 b2,0,b1,000}

movm, r // r => m

movn, n

lsrc, sm, 3'b011 // m= {000,b4,b3,b2,0,b1}

lsrc, sn, 3'b001 // n= {b1,000\_0000}

lsrc, sm, 3'b010 // m= {00000,b4,b3,b2}

lslc, sm, 3'b101 // m= {b4,b3,b2,b1,0000}

movx, d

vall, 4'b0001

valh, 4'b0000

movy, v // {b11,b10,b9,b8,b7,b6,b5,p8} ror {00000001}

mthr, ror // r= {p8,b11,b10,b9,b8,b7,b6,b5}

movn, r // r => n

lsrc, sm, 3'b100 // m= {b8,b7,b6,b5,b4,b3,b2,b1}

movc, m // m => c

movm, m // m= 00000000

lslc, sn, 3'b001 // n = {b11,b10,b9,b8,b7,b6,b5,0}

lsrc, sn, 3'b101 // n = {0000\_0,b11,b10,b9}

movx, n

movy, k

mthr, lor // r= {F1,F0,00\_0,b11,b10,b9}

movd, r // r => d

/\*stor\*/ decr, a

stor, sa, d[2:0]

decr, a

stor, sa, c[2:0]

/\*comp\*/ bnzr, a

func, done

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

/\*sub0\*/ movx, v //subroutine: calc parity using mask in v, store in r

movy, c

mthr, amp // mask c

movy, d

mths, amp // mask d

movx, r

movy, s

mthr, parx // calculate byte parity

mths, pary

movx, r

movy, s

mthr, eor // r= parity {d,c}-mask

func, rfsr

## Program 3 Assembly Code

/\*init\*/ vall, 4'b1111

valh, 4'b0000

movz, v // branch target 15

vall, 4'b0000

valh, 4'b0010

movc, c // c: occurences in byte

movd, d // d: occurences across byte

movk, k // k: bytes containing occurence

movb, v // b= 32 (count down)

mova, a // a= 0 (count up)

load, sb, x[2:0] // x= pattern

load, sa, m[2:0] // m= bit sequence 01234567

incr, a

decr, b

func, noop

/\*load\*/ movj, j // j: occurred in byte

load, sa, n[2:0] // n= bit sequence 89abcdef

incr, a

decr, b

/\*cmp\*/ jtsr, 4'd1

jtsr, 4'd1

jtsr, 4'd1

jtsr, 4'd1

jtsr, 4'd2

jtsr, 4'd2

jtsr, 4'd2

jtsr, 4'd2

movy, j

mths, revy // y = j[0:7]

jizr, ss, 3'b001

incr, k

/\*comp\*/ bnzr, b

/\*last\*/ movj, j

movn, n

jtsr, 4'd1

jtsr, 4'd1

jtsr, 4'd1

jtsr, 4'd1

movy, j

mths, revy // y = j[0:7]

jizr, ss, 3'b001

incr, k

/\*stor\*/ incr, v // v= 33

movb, v // v => b

stor, sb, c[2:0] // mem[33] <= c

incr, b

movm, k

stor, sb, m[2:0] // mem[34] <= m <= k

incr, b

movx, c

movy, d

mthr, add // r= c + d

stor, sb, r[2:0] // mem[35] <= c + d

func, done

func, noop

func, noop

func, noop

func, noop

func, noop

func, noop

/\*sub1\*/ mthr, eql5 // check for in-byte occurrence

jizr, sr, 3'b010

incr, c

incr, j

func, noop

lslc, sm, 3'b001

lslc, sn, 3'b001 // shift in next bit

func, rfsr

func, noop

func, noop

/\*sub2\*/ mthr, eql5 // check for across-byte occurrence

jizr, sr, 3'b001

incr, d

lslc, sm, 3'b001

lslc, sn, 3'b001 // shift in next bit

func, rfsr

# Changelog

* Milestone 3
  + Program Implementation
    - Replaced updated instructions.
    - Incorporated subroutines
    - Added bug fixes. Programs now run correctly.
* Milestone 2
  + Architectural overview
    - Updated diagram.
  + Machine Specification
    - Instruction Format
      * Replaced movp instruction with jtsr to reflect architectural changes.
      * Removed seth instruction b/c I don’t use it in my program.
      * If I have time I may write some kind of push/pop instructions
    - Operations
      * Added missing instructions (everything that wasn’t a math/logic instruction).
    - Internal Operands
      * Changed literal register (l) to value register (v).
      * Removed PC as a register (not allowed).
      * Added link register (l).
      * Added register encoding values.
    - Control Flow
      * Removed the branch always (movp) instruction.
      * Added information on the new jtsr and rfsr instructions.
  + Programmer’s Model
    - 4.1 Added suggestion to use subroutines in code to reduce writing code multiple times.
    - 4.3 Added response.
  + Individual Component Specification
    - Added components.
  + Changelog
    - Added changelog.
* Milestone 1
  + Initial version