Violet Team

Jordan Ansari, Ian Barthel, Logan Bryant, Hunter Masur

**Design Description**

Our processor will follow the accumulator architecture. An accumulator-based architecture is where operations and their results are stored in a single register. This allows for very simple instructions which consist of only an opcode and an address, because the register need not be specified. Our intention is to allow for everything a standard processor would include- this has been proven to be possible, as the original Gameboy ran on an accumulator architecture.

**Registers**

$cr | $cr1, $cr2, $cr3

CR stands for cool register. Although they cannot be explicitly accessed by the programmer, they are the registers they will implicitly access and edit via their instructions. The main $cr will be one of the three listed, and can be changed with the mov command.

$pc

PC is the program counter, which tracks the line being executed.

$sp

SP is the stack pointer, which points to the current location in the stack where memory is being written.

$ra

RA is the return address for any jump command (i.e. where PC should go when a procedure returned).

$t1, $t2

T registers are temporary registers for storing information that is not preserved across function calls.

$a1, $a2

A registers are argument registers used for arguments going into a procedure.

$v1, $v2

Registers used to return arguments from a procedure.

$at

A register used for pseudo-instructions.

$0

The zero register.

**Machine Instructions**

Mov- Swap between the accumulator register being used

Set- Set an accumulator value to a register

Setc- Set a register to an accumulator value

Sw- Store a value at an address in memory

Lw- Load a value from an address in memory

J- Jump to an instruction address from the current location

Beq- Jump to a label if $cr is equal to a value

Bne- Jump to a label if $cr is not equal to a value

Jal- Jump to an address in memory and store the return address (link back)

Sub- Subtract a value from $cr

Add- Add a value to $cr

Addi- Add an immediate to $cr

Ori- Or the binary value of $cr with the binary value of an immediate

Lui- Load an immediate into the upper bits of $cr

Slt- Set $cr to 0 if it is less than the given value, or 1 if it is greater.

Sll- Shift the bits of $cr to the left by a given value

Instructions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction name | Syntax | Behavior | Semantics | Addressing Type | OpCode |
| Mov | Mov $cr\_ | $cr = $cr(n) | Switch current accumulator register to $\_ | Psuedo Direct | 00000 |
| Set | Set $\_ | $cr = $\_ | Set the value of the current accumulator register to $\_ | Direct | 00001 |
| Setc | Setc $\_ | $\_ = $cr | Set the value of $\_ to the current accumulator’s value | Direct | 00010 |
| Sw | Sw $\_ | Mem[$\_] = $cr | Store $cr at address $\_ | Offset | 00011 |
| Lw | Lw $\_ | $cr = Mem[$\_] | Load the value from $\_ into $cr | Offset | 00100 |
| SetPC | SetPC n | $cr = PC + 2 + n | Store $PC+2 + n shifted left by 1 (usually will be 0) | Direct | 00101 |
| GetIn | GetIn $\_ | $\_ = input | Load the outside input into $\_ (selected register in register file) | Direct | 00110 |
| J | J LABEL | $pc = JumpAddr | Jump to the label in the instructions | Psuedo Direct | 100 |
| Beq | Beq LABEL | if($cr == 0) then $pc = pc + 2 + BranchAddr | Jump to the label in instructions if $cr is equal to n | PC relative | 110 |
| Bne | Bne LABEL | if($cr != 0) then $pc = pc + 2 + BranchAddr | Jump to the label in instructions if $cr is not equal to n | PC relative | 111 |
| Sub (R) | Sub $\_ | $cr = $cr – $\_ | Subtract register data from $cr | Direct | 01001 |
| Add(R) | Add $\_ | $cr = $cr + $\_ | Add register to $cr | Direct | 01010 |
| Addi (I) | Addi n | $cr = $cr + immediate | Add the immediate n to $cr | Immediate | 01011 |
| Slt | Slt n | Slt: $cr > $\_ ? $cr = 1 if true, = 0 false | Set $cr to 0 if $cr is less than n, or 1 if it is greater than n | Direct | 01110 |
| Sll (R) | Sll n | Sll: $cr = $cr shifted “left” by immediate (negative values shift right) | Shift the bits of $cr to the left by n | Immediate | 01111 |

Pseudo Instructions

|  |  |
| --- | --- |
| Pseudo Instruction | Actual Instructions |
| Sav | Mov 1  Setc $at  Set $0  Ori 0x00  Lui 0xC0  Swc $at  (Repeat for 2 and 3, with Ori 0x04 and 0x08) |
| Load | Mov 1  Set $0  Ori 0x00  Lui 0xC0  Setc $at  Lw $at  (Repeat for 2 and 3, with Ori 0x04 and 0x08) |
| Subi n | Addi -n |

Instruction Format

R-type:

|  |  |
| --- | --- |
| Op (15-11) | Address(10-0) |

I-type:

|  |  |
| --- | --- |
| Op (15-11) | Immediate(10-0) |

J-type:

|  |  |
| --- | --- |
| Op (15-13) | Address (12-0) |

**Procedure Call Conventions**

$0

Number: 0

Use: Value of 0

Preserved across a call? Yes

$pc

Number: 1

Use: Tracks the program counter

Preserved across a call? Yes

$sp

Number: 2

Use: Stack pointer

Preserved across a call? Yes

$ra

Number: 3

Use: Return address

Preserved across a call? Yes

$t1, $t2

Number 4-5

Use: Temporary registers

Preserved across a call? No

$a1, $a2

Number 6-7

Use: Argument registers

Preserved across a call? No

$v1, $v2

Number 8-9

Use: Return registers

Preserved across a call? No

Register Box

|  |  |  |
| --- | --- | --- |
| Register | Use | Preserved Across Call? |
| $0 (0) | Value is always 0 | Yes |
| $pc (1) | Keeps the PC value | Yes |
| $sp (2) | Stack pointer- keeps track of stack | Yes |
| $ra (3) | Used to store a return address | Yes |
| $t1, $t2 (4-5) | Temporary registers to store data to access | No |
| $a1, $a2 (6-7) | Argument Registers | No |
| $v1, $v2 (8-9) | Return Registers | No |
| $at (10) | Used for pseudo instructions | No |

Cool Register Box

|  |  |  |
| --- | --- | --- |
| $cr1, $cr2, $cr3, $cr4 (0-3) | The accumulator registers that can be switched between | No |

Memory Box

|  |  |
| --- | --- |
| Location | Address |
| Kernel | 0x01F0 |
| Program Start | 0x0000 |
| Cool Registers | 0x03F0 |
| Stack | 0x02F0 |

**Example Program (old)**

0x0000 RELPRIME:

0x0002 mov $cr1 0000000000000100

0x0004 set $0 0000100000000000

0x0006 addi 2 0101100000000010

0x0008 setc $a2 0001000000001010

0x000a GetIn

0x000c setc $a1

0x000a RELPRIMELOOP:

0x000c j GCD 1010100000010101

0x000e BACKHERE:

mov $cr2 0000000000000101

0x0010 set $v1 0000100000001011

0x0012 mov $cr3 0000000000000110

0x0014 set $0 0000100000000000

0x0016 addi 1 0101100000000001

0x0018 slt $v1 0111000000010011

0x001a beq DONERELPRIME 1100100000010010

0x001c mov $cr1 0000000000000100

0x001e addi 1 0101100000000001

0x0020 setc $a1 0001000000001001

0x0022 j RELPRIMELOOP 1000100000000101

0x0024 DONERELPRIME:

mov $cr1 0000000000000100

0x0026 setc $v1 0001000000010011

0x0028 j $ra 100xxxxxxxxxxxxx

0x002a GCD:

mov $cr2 0000000000000101

0x002c set $a1 0000100000001001

0x002e beq ENDB 1100100000101011

0x0030 mov $cr1 0000000000000100

0x0032 set $a2 0000100000001010

0x0034 LOOP:

mov $cr2 0000000000000101

0x0036 beq ENDA 1100100000101110

0x0038 mov $cr1 0000000000000100

0x003a setc $t1 0001000000000111

0x003c mov $cr2 0000000000000101

0x003e setc $t2 0001000000001000

0x0040 mov $cr3 0000000000000110

0x0042 set $t2 0000100000001000

0x0044 slt $t1 0111000000000111

0x0046 beq AGB 1100100000100101

0x0048 j BGA 1000100000101000

0x004a AGB:

mov $cr1 0000000000000100

0x004c sub $a2 0100100000001010

0x004e j LOOP 1000100000011010

0x0050 BGA:

mov $cr2 0000000000000101

0x0052 sub $a1 0100100000001001

0x0054 j LOOP 1000100000011010

0x0056 ENDB:

mov $cr2 0000000000000101

0x0058 setc $v1 0001000000010011

0x005e j BACKHERE 1000100000000111

0x0060 ENDA:

mov $cr1 0000000000000100

0x0062 setc $v2 0001000000001100

0x0064 j BACKHERE 1000100000000111

**Example Program (new)**

RELPRIME: mov $cr1

set $0

addi 2

setc $a2

setc $v2

GetIn

setc $a1

RELPRIMELOOP: j GCD

BACKHERE: mov $cr2

set $v1

mov $cr3

set $0

slt $v1

beq DONERELPRIME

mov $cr1

setc $v2

addi 1

set $v2

setc $a1

j RELPRIMELOOP

DONERELPRIME: mov $cr1

setc $v1

j 1022

GCD: mov $cr1

set $a1

beq ENDB

mov $cr2

set $a2

LOOP: mov $cr2

beq ENDA

mov $cr1

setc $t1

mov $cr2

setc $t

mov $cr3

set $t1

slt $t2

j BGA

AGB: mov $cr1

sub $a2

j LOOP

BGA: mov $cr2

sub $a1

j LOOP

ENDB: mov $cr

setc $v1

j BACKHERE

ENDA: mov $cr1

setc $v2

j BACKHERE

**Common Operation Fragments**

Loading and storing address into a register:

ex: Loading into $t1 (a is at address 0x04)

0x1000 Mov 1 0000000000000001

0x1002 Ori 4 0110011111111100

0x1004 Lwc $t1 0011000000000111

Ex: storing $t1 (at address 0x04)

0x1000 Mov 1 0000000000000001

0x1002 Ori 4 0110011111111100

0x1000 Swc $t1 0010100000000111

Iteration: ex array[0-4] (size 5)

0x1000 Ori 0

0x1002 Sw mem[addr] (store I in a memory address specifically for iteration vars ex: I, j, k)

0x1004 Loop: mem[addr of 0 of array]

0x1006 Add mem[I or iteration var]

0x1008 Lw mem[I]

0x100a Addi immediate (1 or 4 or however the PC instructions increase)

0x100c Sw mem[I]

0x100e bne mem[addr], Loop (Branch not equal to array size (5))

Conditional statement:

0x1000 Bne LOOP – jumps to LOOP if $cr != 0

0x1000 Beq LOOP - jumps to LOOP if $cr == 0

Recursive Statement (Multiplication Function):

0x1000 Mov $cr1 0000000000000100

0x1002 Addi 10 0101100000001010

0x1004 Setc $a1 0001000000001001

0x1006 Mov $cr2 0000000000000101

0x1008 Addi 3 0101100000000011

0x100a Setc $a2 0001000000001010

0x100c jal MULT 1010100000000111

0x100e MULT:

0x1010 Mov $cr1 0000000000000100

0x1012 Set $a1 0000100000001001

0x1014 Addi –1 0101111111111111

0x1016 Beq DONE 1100100000010100

0x1018 Setc $a1 0001000000001001

0x101a Jal MULT 101

0x101c Mov $cr1 0000000000000100

0x101e Set $v1 0000100000001011

0x1020 Add $a2 0101000000001010

0x1022 Setc $v1 0001000000001011

0x1024 J $ra 100

0x1026 DONE:

0x1028 Mov $cr1 0000000000000100

0x102a Set $a2 0000100000001010

0x102c Setc $v1 0001000000001011

0x102e J $ra 100

**RTL Specification**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | R-type | lw/sw | mov | beq/bne | j |
| Inst Fetch | IR = Mem[PC]  PC = PC + 2 | | | | |
| Inst Decode  Register Fetch | Op = IR[15-11]  Adr/Imm = IR[10-0] | | | Op = IR[15-13]  Adr = IR[12-0] PC = PC([15-14]) || Se(Adr<<1) | |
| Execution  Address comp  Instructions done | CoolReg[CurCool] = CR op Reg[Adr] | Lw: CoolReg[CurCool] = Mem[Reg[Adr]]  Sw: Mem[Reg[Adr]] = CRVal | SetID = 1  CRF= Imm | if(CoolReg[CurCool] == 0 [bne: CoolReg[CurCool] != 0]) then  PC = Adr |  |

**Component Specifications**

Component Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Component | Input signals | Output Signals | Control Signals | Description | Test Description |
| IR | Mem Out [PC] (16) | Inst (16) | CLK(1)  IRenable(1) | The instruction will break up the instruction into different busses. | Ensure the output busses have the appropriate bits. |
| ALU | A & B  A and B can come from immediate or register (RegA, RegB) | ALU Out (16) | ALU OP (2 bits) | The ALU will perform whatever computation that is designated by ALU OP on A and B to generate ALU out | Input a variety of values and operations and compare the given output to the desired output |
| Register File  [Reg] | Instruction signals-  Inst[10-0] | RegDat (16) | RegW (1),  CLK(1) | The register file will hold the different cr and other registers to access | Test writing and retrieving data from registers and comparing the values |
| Memory File  [Mem] | Addr = inst[10-0],  RegA | Mem Out (16) | MemW (1)  CLK (1) | The Memory File will hold the memory data to load and store data | Ensure data is pulled in the proper order and is the correct values |
| PC ALU | PC | PC Out = PC + 2 (16) | NA | This will handle the incrementing of the PC as the program progresses | Input a variety of 16-bit numbers to see if they increment correctly |
| SE | Immediate = inst[10-0] | SE immediate (16) | None | Will sign extend the immediate provided for computations | Input 10-bit immediates and compare to the correct value |
| CoolReg File  [CRF] | 2 bits from mov instruction, | CRDat(16) | SetID(1)  CRW(1) | Will be a separate register file that will hold the accumulator registers that are always used for data | Test writing and retrieving data to each register, including when SetID, CRW, and CRR are the wrong values |
| Control Unit | OP code | All control signals. | NA | Will determine all of the signals for the datapath based on the instruction |  |

**Control Signals**

|  |  |  |
| --- | --- | --- |
| **Signal** | **Bits** | **Description** |
| SetID | 1 | If the SetID is 1, then the cool register is allowed to switch the CurCool register used for accumulator register calculations |
| ALU OP | ... | Controls the operation taken by the ALU: addition, subtraction, and, or, etc |
| CRW | 1 | CRW enables writing data to the coolRegister that the ID is set to |
| MemW | 1 | MemW enables writing data to Memory |
| MemR | 1 | MemR enables reading data from Memory |
| RegW | 1 | RegW enables writing data to the Register File with timep registers, argument registers, etc. |
| ASel | 2 | Selects which input goes into the ALU as the A: can be data from RegFile, CoolReg File |
| BSel | 2 | Selects which input goes into the ALU as the B: can be data from RegFile, CoolReg File, Sign extended immediate, or 0 |
| CLK | 1 | Clock register which switches from 0 to 1 to 0 (ad inf.), and causes updates on the “rising edge,” or when set equal to 1. |
| IRenable | 1 | Enables writing the instruction from memory at PC |
| IorD | 1 | IorD is used to select between ALUOut and the sign-extended instruction when writing into memory |
| SetID | 1 | SetID enables the swapping of Cool Registers |
| PCSource | 2 | Selects the input to go into the PC |

**System Testing Plan**

Step 1: Test simple machine instruction, one for each instruction

Mov 2 0000000000000010

Addi 4 0101111111111100

Set $t1 0000100000000111

Sw $t1 0001100000000111

Sub $t1 0100100000000111

Ori 0x02 0110011111111110

Lui 0x04 0110111111111100

Lw $t1 0010000000000111

SetC $t1 0001000000000111

Add $t1 0101000000000111

Swc $t1 0010100000000111

Lwc $t1 0011000000000111

J 0x1004 1001000000000100

Beq 0x1004 1101000000000100

Bne 0x1004 1111000000000100

Jal 0x1004 1011000000000100

Slt 5 0111000000000101

Sll 2 0111100000000010

Step 2: Test code fragments (reference code fragments)

* Storing into a register
* Iteration

Step 3: Test Euclid's algorithm (reference example program)

**Bottom-Up Testing Plan**

Step 1: Implement and test each individual component: IR, ALU, Reg File, Mem File, PC ALU, Sign Extender, CR file, ALU Out, Control Unit

Step 2: Implement and test subsystems based on the datapath’s cycles.

* System 1: IR, Reg File, Cool Reg File, SE,
  + The biggest part of our datapath, test whether the register file can correctly pull data from inside a register using info from the IR.
* System 2: ALU (with A and B selectors)
  + Test whether the ALU can perform all necessary operations, given any of our 8 total inputs for A and B.
* System 3: ALU Out, mem file, PC
  + Test that mem file pulls the correct information from memory along with being able to read ALU Out and combine with PC to perform jumps.
* System 4: Control Unit

Step 3: Combine each subsystem in the datapath from left to right, testing after each subsystem is added.

* Addition 1: System 1,2
* Addition 2: System 1,2,3 (datapath)
* Addition 3: System 1,2,3,4 (datapath with controls)

**Component Testing Plan (with examples)**

**IR**: Input instructions and compare the output buss(es) to their expected values

Test 1:

Input: 1111 1111 1000 0000

Expected output: 111 1000 0000

Test 2:

Input: 0000 0000 1111 1111

Expected output: 000 1111 1111

**ALU**: Test ADD, SUB, and SLL by inputting the correct op code and two values and compare the result to the expected value.

Test 1 (add):

Input A: 0000 0000 0000 0011

Input B: 0001 0000 0000 0000

Op: 00

Expected ALU Out: 0001 0000 0000 0011

Test 2 (sub):

Input A: 0000 0001 1111 0000

Input B: 0000 0001 1100 0000

Op: 01

Expected ALU Out: 0000 0000 0011 0000

Test 3 (sll):

Input A: 0000 0001 0011 0000

Input B: 10

Op: 10

Expected ALU Out: 0000 0100 1100 0000

**Register File**: Test that when inputting each registers value that register can both have data input into it and data output from it.

Test 1 (writing):

Reg: 0111 ($t1)

Write Data: 0000 1111 1111 1111

Expected Read Data: 0000 1111 1111 1111

Test 2 (just reading): note: will need to run test 1 to store a value first

Reg: 0111 ($t1)

Write Data: xxx

Expected Read Data: 0000 1111 1111 1111

**Memory File**: Test writing data into memory by attempting to write data to a point in memory and then comparing the value at that point to the data originally written into it.

**TODO: Revise mem tests (has potential issues)**

Test 1: input addr output inst starts from 0x1000

Addr: 0000 0000 0000 1000 (has instruction addi 5)

Write Data: xxxx

Expected MemData: 0101 1000 0000 0101 (addi 5)

Test 2:

Addr: 0000 0000 0000 1000 (addi 5)

Write Data: 0000 0000 0000 1010 (sub 2)

Expected MemData: 0100 1000 0000 0010 (sub 2)

**PC ALU**: Test that the PC counter increases by 2 when moving to the next instruction and test that the PC counter changes to an address when given a jump instruction.

Test 1:

PC (old): 0000 0000 0000 0000

Expected PC (new): 0000 0000 0000 0010

Test 2:

PC (old): 0000 0000 0000 1110

Expected PC (new): 0000 0000 0001 0000

**Sign Extender**: Input positive and negative numbers and compare the output to the expected value of the number when sign extended.

Test 1:

Immediate: 110 1001 1111

Expected SE Immediate: 1111 1110 1001 1111

Test 2:

Immediate: 000 0111 0111

Expected SE Immediate: 0000 0000 0111 0111

**Cool Reg File**: Test that when inputting each registers value that register can both have data input into it and data output from it.

Test 1 (writing):

Inst (cool): 1

Write Data: 0000 1111 1111 1111

Expected Read Data: 0000 1111 1111 1111

Test 2 (reading): note: will need to run test 1 to store a value first

Inst (cool): 1

Write Data: xxx

Expected Read Data: 0000 1111 1111 1111

**Control Unit**: Input each opcode and compare the output signals to the expected signals for that instruction.

Test 1 (mov):

Din: 00000

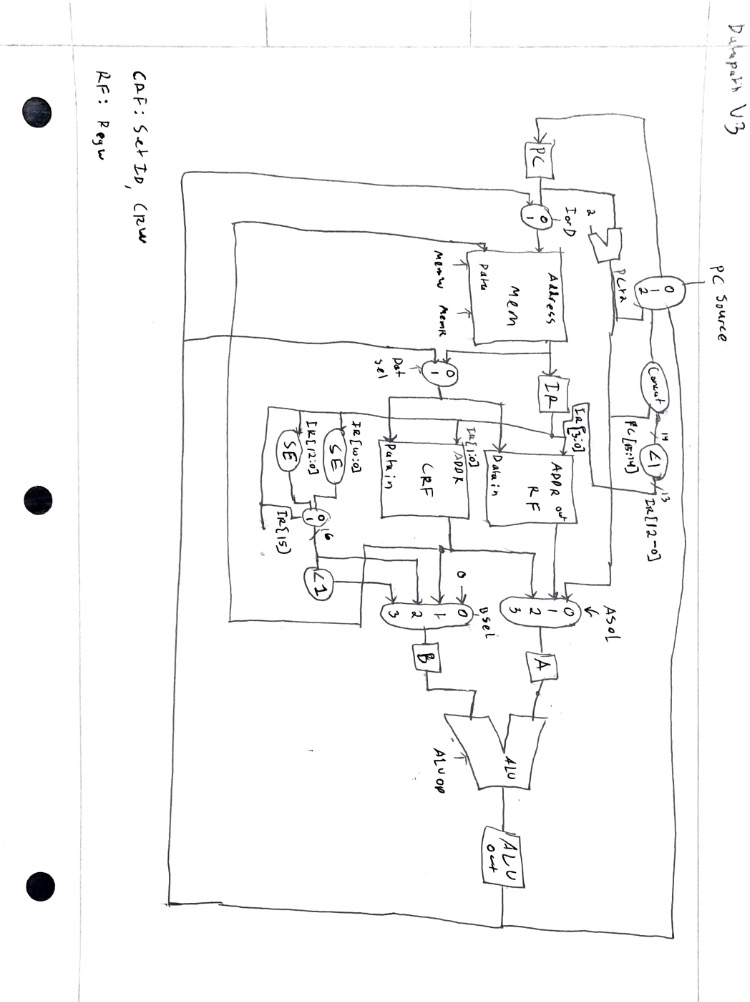
Expected Output: MemRead = 1, wait 1 clock cycle, SetID = 1

Test 2 (lw):

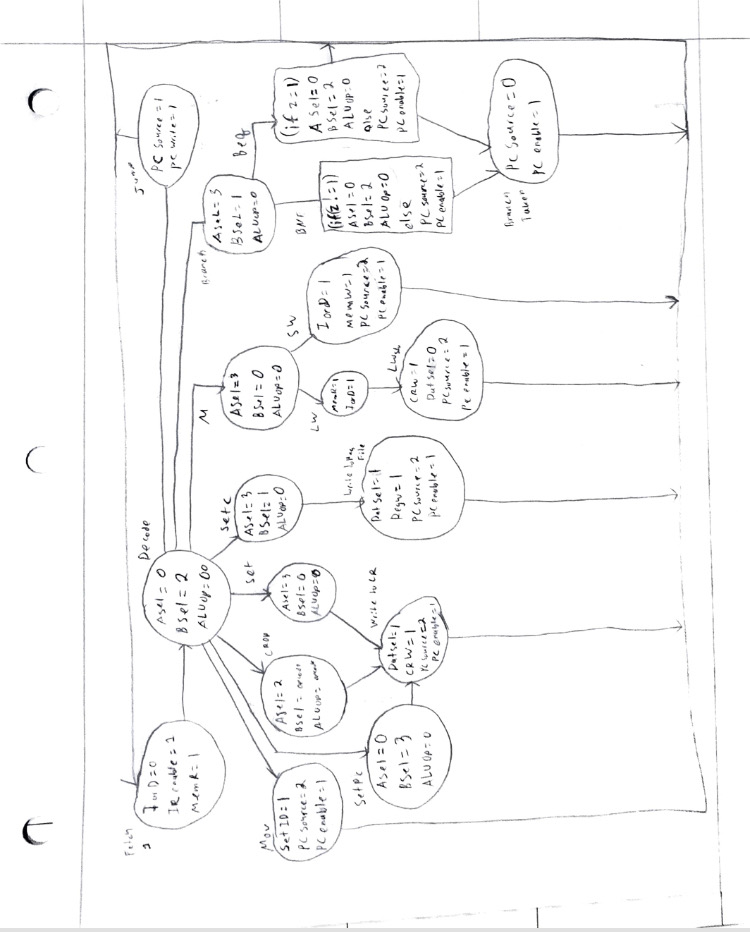
Din: 00100

Expected Output: Memread = 1 | ASel = 01, BSel = 01, IorD = 1 | MemRead = 1 | DataSel = 1, RegW = 1

**Datapath**



**State Diagram**



**Patch Notes**

Milestone 2:

* Changed beq/bne to work with 16 bit instructions: Now only checks if $cr is equal to 0
  + J type instructions were made to have 3 bits in front and 13 bit immediate/branch
  + The first bit (most significant) will be a 1 if the instruction is a J type
* Fixed opcodes to be 5 bits
* Added RTL table
* Added component specifications
* Added component table to specify each components specific signals
* Placed register information in a table
* Added recursive code snippet
* Changed opcodes so that J functions begin with 1, and the second 2 bits are part of the opcode, allowing for longer addresses
* Added mov to the RTL
* Added a cool register file to separate the accumulator registers from the register file
* Added pseudoinstructions

Milestone 3:

* Updated components to add bits to controls
* Added a table with control signals with descriptions
* Added a bottom-up testing plan and details for tests
* Added a component testing plan
* Added control signals for instructions

Milestone 5:

* Updated Subsystems
  + Combined subsystems 1 and 2, except for PC
  + Moved PC into system 4
  + Moved ALU into system 1, removed system 3
* Added datapath scan
* Switched control signal plan to State Machine

Notes:

* 3-bit opcode and 2-bit funct instead of two different opcodes
* Add # of bits for component table
* Add missing inputs to component table (ex: Clock input)
* Map symbols from RTL to Component Specification (be explicit)
* Be careful about J vs other instructions- if the mux for adr is wired to control, it won’t work- if it’s wired to bit 15 of IR, it will probably work as long as adr doesn’t need to be updated quickly
* Instruction is lost every cycle (add enabler input)
* PC and Stack grows up
* input and output:
* add number of bits to signals
* missing some inputs ex clock
* symbol map in component Specifications
* Address code into memory it’ll take time
* CTL and Adr change at same time, but wiring directly to opcode would probably work fine as long as does not need to work immediately (for the opcode of J vs other instructions)