Muffin Rockets Processor

Table of Contents

Registers

- Reserved (Hardwired) Registers
    - $0 - Register 0 is hardwired to 0 $0.
- Argument Registers
    - $1 - Register 1 is meant for argument $a0.
    - $2 - Register 2 is meant for argument $a1.
- Temporary Registers
    - $3 - Register 3 is meant for temporary $t0.
    - $4 - Register 4 is meant for temporary $t1.
    - $5 - Register 5 is meant for temporary $t2.
    - $6 - Register 6 is meant for temporary $t3.
- Save Registers
    - $7 - Register 7 is meant for save value $s0.
    - $8 - Register 8 is meant for save value $s1.
    - $9 - Register 9 is meant for save value $s2.
- Stack Pointer
    - $10 - Register 10 is meant for the stack pointer $sp.
- Return Address
    - $11 - Register 11 is meant for the return address $ra.
- Return Value
    - $12 - Register 12 is meant for the return value $rv.
- Assembler Temporary
    - $13 - Register 13 is meant for an assembler temporary value $at.
- Interrupt Handlers
    - $14 - Register 14 is meant for interrupt handling $i0.
    - $15 - Register 15 is meant for interrupt handling $i1.

Assembly Conventions

- Callers are required to:
  - Create a stack to save their current return address.
  - Fill registers 2 and 3 with arguments meant for the procedure being called.
  - Fill registers 10-12 with values meant to persist through the procedure call.
  - Update the return address register with the address of the instruction after the jump to the procedure call.
  - Jump to the procedure call.
  - Load values from the stack into the corresponding register.
    1. Specifically, the return address should be put into register 11.
  - Undo the stack.
  - If relevant, use register 12 as the return value of the procedure call.
- Callees are required to:
  - Save values in the save registers to memory.
    1. Note, callees are free to use temporary registers.
  - Load the save values from memory to registers before returning.
    1. These will not be modified throughout the rest of the procedure.
  - Use register 12 as storage for a single return value.
  - Jump to the return address.
- For Interrupt handling, two non-clear warp instructions should be separated with a single no-op (add $0 $0 $0).
- If branch is out of range, an inverse conditional branch should branch over a jump instruction. For example, if a branch intended to go forward 8 instructions, the following should be used instead:

```
bne    ___    ___    # Branch out of range for beq.
j      ___    ___    # Address to jump to.
                     # Line bne should jump to.
```

Assembly to Machine Translation

- Opcode will denote the bits that will determine if the instruction is an add, load word, save word, branch, etc.
- Rs will denote the source register for operations whose operands are registers but will not change the value in rs.
- Rt will denote the secondary register for operations whose operands are registers but will not change the value in rt.
- Rd will denote the destination register for operations where the result will be stored in the destination register, overwriting the value that was there.
- Immediate will denote an integer value.

- Instruction Types
  - A-Type

4-Bit Opcode	4-Bit Register Source	4-Bit Register Temporary	4-Bit Register Destination

  - M-Type

4-Bit Opcode	4-Bit Register Source	4-Bit Register Destination	4-Bit Immediate

  - C-Type

4-Bit Opcode	4-Bit Register Destination	8-Bit Immediate

  - J-Type

4-Bit Opcode	12-Bit Immediate

  - IH-Type

0x0101	4-Bit Register Source	4-bit Selector	4-Bit Function Code

Instruction Details

Op code	Instruction	Instruction name	Type	Instruction format
0000	add	Add	A	$rd, $rs, $rt | $rd = $rs + $rt
0001	sub	Subtract	A	$rd, $rs, $rt | $rd = $rs - $rt
0010	and	And	A	$rd, $rs, $rt | $rd = $rs & $rt
0011	or	Or	A	$rd, $rs, $rt | $rd = $rs | $rt
0100	addi	Add immediate	C	$rd, IMM | $rd = $rd + IMM
0101	warp	Warp	IH	Varies (see below)
0110	ori	Or immediate	C	$rd, IMM | $rd = $rd | IMM
0111	slt	Set less than	A	$rd, $rs, $rt | $rd = 1 if ($rs < $rt), 0 otherwise
1000	lw	Load word	M	$rd, $rs, IMM | $rd = Mem[$rs + IMM]
1001	sw	Store word	M	$rd, $rs, IMM \| Mem[$rs + IMM] = $rd
1010	lui	Load upper immediate	C	$rd, IMM | $rd = IMM(upper 8 bits)
1011	beq	Branch equal	M	$rs, $rt, IMM | PC = PC + IMM if $rs = $rt.
1100	bne	Branch not equal	M	$rs, $rt, IMM | PC = PC + IMM if $rs != $rt.
1101	jal	Jump and link	J	IMM | $ra = PC + 1 | PC = PC[15:12] + IMM
1110	j	Jump	J	IMM | PC = PC[15:12] + IMM
1111	jr	Jump register	J	$rs | PC = $rs

Warp Function Details

Warp function	Function code	Instruction	Instruction format
Move	0000	mv	$rs xxxx 0000
Clear	0001	clr	xxxx selector 0001
Read	0010	rd	xxxx xxxx 0010
Display	0011	dsp	xxxx location 0011

Instruction Formats

- Add
  - add $rd, $rs, $rt | $rd = $rs + $rt

0x0 (4 bits)	rs (4 bits)	rt (4 bits)	rd (4 bits)

- Subtract
  - sub $rd, $rs, $rt | $rd = $rs - $rt

0x1 (4 bits)	rs (4 bits)	rt (4 bits)	rd (4 bits)

- And
  - and $rd, $rs, $rt | $rd = $rs and $rt

0x2 (4 bits)	rs (4 bits)	rt (4 bits)	rd (4 bits)

- Or
  - or $rd, $rs, $rt | $rd = $rs or $rt

0x3 (4 bits)	rs (4 bits)	rt (4 bits)	rd (4 bits)

- Add immediate
  - addi $rd, IMM | $rd = $rd + IMM.

0x4 (4 bits)	rd (4 bits)	immediate (8 bits)

- Move
  - mv $rd | $rd = interrupt_register

0x5 (4 bits)	rd (4 bits)	4-bit unused	0x0

- Clear
  - clr IMM | interrupt_register[IMM] = 0

0x5 (4 bits)	4 bits unused	4-bit immediate	0x1

- Read
  - rd IMM | interrupt_register[IMM] = 0

0x5 (4 bits)	4 bits unused	4-bit immediate	0x2

- Display
  - dsp IMM | displayOut = Mem[decoder[IMM]]

0x5 (4 bits)	4 bits unused	4-bit immediate	0x3

- Or immediate
  - ori $rd, IMM | $rd = $rd or IMM.

0x6 (4 bits)	rd (4 bits)	immediate (8 bits)

- Set less than
  - slt $rd, $rs, $rt | $rd = 1 if ($rs < $rt), 0 otherwise.

0x7 (4 bits)	rs (4 bits)	rt (4 bits)	rd (4 bits)

- Load word
  - lw $rd, $rs, IMM | $rd = Mem[$rs + IMM]

0x8 (4 bits)	rs (4 bits)	rd (4 bits)	immediate (4 bits)

- Store word
  - sw $rd, $rs, IMM | Mem[$rs + IMM] = $rd

0x9 (4 bits)	rs (4 bits)	rd (4 bits)	immediate(4 bits)

- Load upper immediate
  - lui $rd, IMM | $rd = IMM(upper 8 bits)

0xA (4 bits)	rd (4 bits)	immediate (8 bits)

- Branch equal
  - beq $rs, $rt, IMM | PC = PC + IMM if $rs = $rt.

0xC (4 bits)	rs (4 bits)	rt (4 bits)	immediate (4 bits)

- Branch not equal
  - bne $rs, $rt, IMM | PC = PC + imm if $rs != $rt.

0xD (4 bits)	rs (4 bits)	rt (4 bits)	immediate (4 bits)

- Jump
  - j IMM | PC = PC[15:12] + IMM

0xE (4 bits)	immediate (12 bits)

- Jump and link
  - jal IMM | $ra = PC + 1 | PC = PC[15:12] + IMM

0xB (4 bits)	immediate (12 bits)

- Jump register
  - jr $rs | PC = $rs

0xF (4 bits)	rd (4 bits)	0x0 (4 bits)	0x0 (4 bits)

Euclid's Algorithm : Assembler Code

Addr	Instruction		Comments
			# relPrime starts
0x0001	addi	$sp, -3	# Increase the $sp by 3
			# (create 2-word stack).
0x0002	sw	$ra, $sp, 0	# Store the return address onto
			# the stack at $sp[0].
0x0003	sw	$a0, $sp, 2	# Store *n* onto the stack at $sp[2].
0x0004	lw	$a0, $sp, 2	# Load n off of stack.
0x0005	sw	$a1, $sp, 1	# Store *m* at $sp[1].
			# Call GCD.
0x0006	jal	0x010	# Jump to gcd. PC = 0x0010.
0x0007	lw	$a1, $sp, 1	# Load the previous *m* into $a1.
0x0008	addi	$a1, 1	# Add 1 to $a1
0x0009	addi	$rv, -1	# Subtract one from $rv.
0x000a	bne	$rv, $0, 0x9	# Check if $rv = 0; if not, return to 0x0004
0x000b	add	$rv, $0, $a1	# Put the current *m* into the $rv
0x000c	addi	$rv, -1	# subtract one from $rv
0x000d	lw	$ra, $sp, 0	# Restore $ra from stack.
0x000e	addi	$sp, 3	# Undo the stack
0x000f	jr	$ra	# Return to caller function.
			# GCD starts.
0x0010	bne	$a0, $0, 0x1	# If $a0 == 0, branch ahead
			# to 0x0012. Let $a0 = a.
0x0011	j	0x01b	# Assist branch in jumping.
0x0012	bne	$a1, $0, 0x1	# If $a1 == 0, branch ahead
			# to 0x0014. Let $a1 = b.
0x0013	j	0x01d	# Assist branch in jumping.
0x0014	beq	$a1, $a0, 0x4	# Branch if $a1 and $a0 are the same
0x0015	slt	$t0, $a1, $a0	# If b < a, $t0 = 1. Otherwise, $t0 = 0.
0x0016	beq	$t0, $0, 0x2	# If $t0 == 1, branch ahead to 0x0180.
0x0017	sub	$a0, $a0, $a1	# a = a - b.
0x0018	j	0x010	# Jump back to 0x0010.
0x0019	sub	$a1, $a1, $a0	# b = b - a.
0x001a	j	0x010	# Jump back to 0x0010.
0x001b	add	$rv, $a1, $0	# Put b into $rv.
0x001c	jr	$ra	# Return to caller function.
0x001d	add	$rv, $a0, $0	# Set $rv = a.
0x001e	jr	$ra	# Return to caller function.

Euclid's Algorithm : Machine Code

Addr	Machine Code	Assembly Code
0x0000	0100 1010 1111 1101	# addi $sp -3
0x0002	1001 1011 1010 0000	# sw $ra $sp 0
0x0003	1001 0001 1010 0010	# sw $a0 $sp 2
0x0004	1000 0001 1010 0010	# lw $a0 $sp 2
0x0006	1001 0010 1010 0001	# sw $a1 $sp 1
0x0008	1101 0000 0001 0000	# jal 0x010
0x0009	1000 0010 1010 0001	# lw $a1 $sp 1
0x0000	0100 0010 0000 0000	# addi $a1 1
0x000a	0100 1100 1111 1111	# addi $rv -1
0x000b	1100 1100 0000 1000	# bne $rv $0 0x8
0x000c	0000 1100 0000 0010	# add $rv $0 $a1
0x0000	0100 1100 1111 1111	# addi $rv -1
0x000d	1000 1011 1010 0000	# lw $ra $sp 0
0x000e	0100 1010 0000 0011	# addi $sp 3
0x000f	1111 1011 xxxx xxxx	# jr $ra
0x0010	1100 0001 0000 0001	# bne $a0 $0 0x1
0x0011	1110 0000 0001 1011	# j 0x01b
0x0012	1100 0010 0000 0001	# bne $a1 $0 0x1
0x0013	1110 0000 0001 1101	# j 0x01d
0x0000	1011 0010 0001 0100	# beq $a1 $a0 0x04
0x0014	0111 0011 0010 0001	# slt $t0 $a1 $a0
0x0015	1011 0011 0000 0010	# beq $t0 $0 0x2
0x0016	0001 0001 0001 0010	# sub $a0 $a0 $a1
0x0017	1110 0000 0001 0000	# j 0x010
0x0018	0001 0010 0010 0001	# sub $a1 $a1 $a0
0x0019	1110 0000 0001 0000	# j 0x010
0x001a	0000 1100 0010 0000	# add $rv $a1 $0
0x001b	1111 1011 xxxx xxxx	# jr $ra
0x001c	0000 1100 0001 0000	# add $rv $a0 $0
0x001d	1111 1011 xxxx xxxx	# jr $ra

Loading Large Immediates : Assembly Code

Addr	Instruction		Comments
0x0000	lui	$t0, 0x12	# Load 0x12 to upper half of $t0
0x0001	ori	$t0, 0x34	# Load 0x34 to lower half of $t0

Loading Large Immediates : Machine Code

Addr	Machine Code				Assembly Code		
0x0000	1010	0011	0001	0010	# lui	$t0,	0x12
0x0001	0110	0011	0011	0100	# ori	$t0,	0x34

Code Snippets for Other Instructions

```
and     $t2, $t0, $t1       # Let $t0 = 0xf0f0, $t1 = 0xa0a0
                            #         1111   0000   1111   0000
                            #  &      1010   0000   1010   0000
                            #         1010   0000   1010   0000
                            # $t2 = 0xa0a0.

or      $t2, $t0, $t1       # Let $t0 = 0xf0f0, $t1 = 0xafa0
                            #         1111   0000   1111   0000
                            # [or]    1010   1111   1010   0000
                            #         1111   1111   1111   0000
                            # $t2 = 0xfff0

ori     $t0,    0x4a        # Let $t0 = 0xf0f0, imm = 0xfa
                            # imm sign-extended = 0xfffa
                            #         1111   0000   1111   0000
                            # [or]    1111   1111   1111   1010
                            #         1111   1111   1111   1010
                            # $t0 = 0xfffa

lui     $t0,    imm[up]     # 16 bit immediate imm = 0x1234
                            # Load the top 8 bits of imm.
                            # $t0 = 0x12
```

Interrupts

This processor utilizes a special interrupt register.

Its 8 bits are as follows:

7	6	5	4	3	2	1	0
SW3	SW2	SW1	SW0	N	E	S	W

SW = Switch on FPGA
N/E/S/W = North/East/South/West buttons on FPGA

There is a check made in the instruction fetch cycle of each instruction to see if there are currently any un-handled interrupts.

Register Transfer Language

A-type instructions	C-type instructions	Load word/Store word	Branches	Jump	Jump Register	Jump and Link	Load Upper Immediate	Jump ‘n’ Link
I = Mem[PC] PC = PC + 1								
A = Reg[I[11:8]] B = Reg[I[7:4]] ALUOut = PC + sign-extend*(I[3:0])								
ALUOut = A op B	ALUOut = A op sign-extend**(I[7:0])	ALUOut = A + sign-extend*(I[3:0])	if (A == B): PC = ALUOut	PC = PC[15:12] || I[11:0]	PC = Reg[I[11:8]]	ALUOut= PC + 0	Reg[I[11:8]] = extend(I[7:0])	PC = A
Reg[I[3:0]] = ALUOut	Reg[I[11:8]] = ALUOut	load: MDR = Mem[ALUOut] store: Mem[ALUOut] = Reg[I[7:4]]				PC = PC[15:12] || (I[11:0]) Reg[0xb] = ALUOut		
		load: Reg[I[7:4]] = MDR store: no op						

sign-extend* will sign-extend a 4-bit integer to a 16-bit integer.

sign-extend** will sign-extend an 8-bit integer to a 16-bit integer.

sign-extend*** will sign-extend a 12-bit integer to a 16-bit integer .

extend will create a 16-bit value where the most significant 8-bits are the given immediate and the least significant are 0s.

Warp Function RTL

mv	clear	read	display
I = Mem[PC] PC = PC + 1 A = Reg[I[11:8]] B = Reg[I[7:4]] ALUOut = PC + sign-extend*(I[3:0])			
Reg[IR[11:8]]= InterruptRegister	PC = EPC InterruptRegister[SELECT] = 0	Mem[decodeOut] = lcdData	Mem[decodeOut]
		blank	lcdDisplay = Mem[decodeOut]
			blank

interrupt = 1
EPC = PC PC = Int_handler_master_address

Component Design and Description

- PC Register
    - Description: This register will hold the program counter.
    - Input: 16-bit
    - Output: 16-bit
    - Control Signal: PCWrite will indicate if the input will be written to the PC register.
    - Tests:
        - init: input = 0x0000, PCWrite = 1
        - expect: output = 0x0000
        - init: input = 0x0001, PCWrite = 0
        - expect: output = 0x0000
- Memory Block:
    - Description: Block of memory that will store both instructions and data.
    - Input: 16-bit Address whose data will be outputted, 16-bit Data that will be written to the given Address if MemWrite is 1.
    - Output: 16-bit Value at the input Address
    - Control Signal: 1-bit MemWrite will write the given Data to the given Address
    - Tests:
        - init: addr = 0x0000, data = 0x0001, MemWrite = 1
        - expect: read = 0x0000
        - init: addr = 0x0000, data = 0x0002, MemWrite = 0
        - expect: read = 0x0001
        - init: addr = 0x0000, data = 0xffff, MemWrite = 0
        - expect: read = 0x0001 (?)
        - init: addr = 0x0000, data = 0xffff, MemWrite = 1
        - expect: read = 0x0001 (?)
        - init: addr = 0x0000, data = 0x0000, MemWrite = 0
        - expect: read = 0xffff

- Instruction Register:
    - Description: Register to store the current instruction.
    - Input: 16-bit Instruction
    - Output: 4-bit Instruction[15:12], 4-bit Instruction[11:8], 4-bit Instruction[7:4], 8-bit Instruction[7:0]
    - Control Signal: 1-bit RegWrite will store the incoming instruction to the register.
    - Tests:
        - init: in = 0x1234, RegWrite = 1
        - expect: Read1 = 0x1, Read2 = 0x2, Read3 = 0x3, Read4 = 0x34
        - init: in = 0x4321, RegWrite = 0
        - expect: Read1 = 0x1, Read2 = 0x2, Read3 = 0x3, Read4 = 0x34
- Sign-Extension (12 to 16 bit):
    - Description: Sign-extend the given 12-bit value to a 16-bit value.
    - Input: 12-bit value to be extended.
    - Output: 16-bit value
    - Tests:
        - init: in = 0x123
        - expect: out = 0x0123
        - init: in = 0xf23
        - expect: out = 0xff23
- Register Write Address Mux:
    - Description: 4 Options for the wire whose value will be the register to write to.
    - Input: 4-bit Instruction[11:8], 4-bit Instruction[7:4], 4-bit Instruction[3:0], 4-bit 0xb
    - Output: 4-bit register address
    - Control Signal: 2-bit RegWriteAddr will choose the register to write to.
    - Tests:
        - init: in1 = 0x1, in2 = 0x2, in3 = 0x3, in4 = 0xb, RegWriteAddr = 0x0
        - expect: out = 0x1
        - init: RegWriteAddr = 0x1
        - expect: out = 0x2
        - init: RegWriteAddr = 0x2
        - expect: out = 0x3
        - init: RegWriteAddr = 0x3
        - expect: out = 0xb

- Register Write Data Mux:
    - Description: 4 Options for the data to write to a general register.
    - Input: 16-bit ALUOut, 16-bit MDR, 16-bit Extended Instruction[7:0], Interrupt Register
    - Output: 16-bit value
    - Control Signal: 2-bit RegWriteData will choose the data to write.
    - Tests:
        - init: in1 = 0x0001, in2 = 0x0002, in3 = 0x0003, RegWriteData= 0x0
        - expect: out = 0x0001
        - init: RegWriteData = 0x1
        - expect: out = 0x0002
        - init: RegWriteData = 0x2
        - expect: out = 0x0003
- Sign-Extension (8 to 16 bit):
    - Description: Sign-extend an 8-bit value to 16-bits
    - Input: 8-bit value
    - Output: 16-bit value
    - Tests:
        - init: in = 0x12
        - expect: out = 0x0012
        - init: in = 0xf2
        - expect: out = 0xfff2
- Sign-Extension (4 to 16 bit):
    - Description: Sign-extend a 4-bit value to 16-bits
    - Input: 4-bit value
    - Output: 16-bit value
    - Tests:
        - init: in = 0x1
        - expect: out = 0x0001
        - init: in = 0x8
        - expect: out = 0xfff8

- Load-Upper Extension (8 to 16 bit):
    - Description: Extend an 8-bit value to 16 bit value as such:
        - Given 0x12 -> 0x1200
    - Input: 8-bit value
    - Output: 16-bit value
    - Tests:
        - init: in = 0x12
        - expect: out = 0x1200
        - init: in = 0xff
        - expect: out = 0xff00
- Extended Value Mux:
    - Description: Choose which of the sign-extended values to be output.
    - Input: 16-bit value, 16-bit value
    - Output: 16-bit value
    - Control Signal: 1-bit ExtValue will choose the extended value to output.
    - Tests:
        - init: in1 = 0xfff2, in2 = 0x0012, ExtValue = 0x0
        - expect: out = 0xfff2
        - init: ExtValue = 0x1
        - expect: out = 0x0012
- General Register File:
    - Description: Register file for our 16 general registers.
    - Input: 4-bit Read Address 1, 4-bit Read Address 2, 4-bit Write Address, 16-bit Write Data
    - Output: 16-bit Read Out 1, 16-bit Read Out 2
    - Control Signal: 1-bit RegWrite will indicate that we will write the Write Data to the Write Address
    - Tests:
        - init: ReadAddr1 = 0xf, ReadAddr2 = 0xa, WriteAddr = 0xc, Write = 0x1234, RegWrite = 1
        - expect: Read1 = (initial value), Read2 = (initial value)
        - init: ReadAddr1 = 0xc, ReadAddr2 = 0xc, WriteAddr = 0xf, Write = 0x1234, RegWrite = 0
        - expect: Read1 = 0x1234, Read2 = 0x1234
        - init: ReadAddr1 = 0xf, ReadAddr2 = 0xf, WriteAddr = 0xb, Write = 0x4212, RegWrite = 1
        - expect: Read1 = not 0x1234, Read2 = not 0x1234

- A Register:
    - Description: Register to store the Read Out 1 value from General Reg. File
    - Input: 16-bit Read Out 1
    - Output: 16-bit value
    - Tests:
        - init: in = 0x1234
        - expect: out = 0x1234
- B Register:
    - Description: Register to store the Read Out 2 value from General Reg. File
    - Input: 16-bit Read Out 2
    - Output: 16-bit value
    - Tests:
        - init: in = 0x1234
        - expect: out = 0x1234
- ALU Source A Mux:
    - Description: The unit for deciding what will go into the first half of the ALU
    - Input: It has two inputs that are 16-bit values. The inputs include what is in register A and the value of the PC register.
    - Output: It will have an output of a 16-value.
    - Control Signal: It is given a control signal that is ALUSrcA that tells which to use.
    - Tests: We will write tests for the different inputs.
    - Inputs: 0x0003 in 0, 0x0445 in 1. If mux is 0, output expected is 0x0003. If mux is 1, 0x0445 is expected output.
        - add $t0, $t0, $t1 - $t0 has value 0x00aa and $t1 has value 0x0011
            - We expect the new value 0x00bb to be stored into register $t0 and the program counter to be 1 higher.
        - sw $t0, ($a0)0 - $a0 has a value of 2 and $t0 is at memory 0x002a
            - We expect that 2 will be stored at the location 0x002a

- ALU Source B Mux:
    - Description: The unit for deciding what will go into the second half of the ALU
    - Input: It has four inputs that are 16-bit values. The inputs include what is in register B, an immediate of a 1, different immediates that were sign-extended.
    - Output: It will have an output of a 16-value.
    - Control Signal: It is given a control signal that is ALUSrcB that tells which to use.
    - Tests: We will check the 4 different settings of the mux
    - Inputs: 0x002a, 0x0001, 0x00ff, 0x0fff. If mux is 0, output is 0x002a. If mux is 1, output is 0x0001. If mux is 2, output is 0x00ff. If mux is 3, output is 0x0fff.
        - jal $t0 - $t0 has value 0x0004 and we are at 0x0001
            - We expect the value in PC register to change to 0x0004 and the value of the line after 0x0001 : 0x0002 will be in the $ra register.
        - add $t0, $t0, $t1 - $t0 value 0x00aa and $t1 has value 0x0011
            - We expect the new value 0x00bb to be stored into register $t0 and we expect the program counter to be 1 higher.
        - beq $t0, $t0, 0x0003
            - We expect the program counter to increase by 4. (once at the very beginning and three more when the branch is run.
        - j $t2 - the value of the PC is 0x0002 and $t2 is 0x0ff0
            - We expect the new value of the PC Register to be 0x0ff0
- ALU:
    - Description: The unit for calculation within the processor. This takes care of all the arithmetic computations.
    - Input: It has two inputs that are 16-bit values.
    - Output: It will have an output of a 16-value.
    - Control Signal: It is given a control signal to dictate what operation the ALU will use.

- ALU Operation:
    - Description: This is the operation that we tell the ALU to use.
    - Input: It has one input of a 4 bit value which is taken from the op code of the instruction.
    - Output: It will have an output of a 3-bit value.
    - Tests: We will check many different operations.
    - If it is an add and inputs 0x00dd and inputs 0x0011, we expect the output to be 0x00ee. If it is a subtract, we expect the output to be 0x00cc. If it is an “and”, we expect the output to be 0x0000, and if it is an “or” we expect 0x00ff.
        - add $t0, $t0, $t1 - $t0 value 0x00aa and $t1 has value 0x0011
            - we expect the new value 0x00bb to be stored into register $t0
        - sub $t0, $t1, $t2 - $t0 has a value of 0x0fff, $t1 has value 0x2222, $t2 has value 0x1111
            - We expect that $t0 will now have the value 0x1111
        - or $t1, $t0, $a0 - $t1 has value 0x0137, $t0 has value 0xf00f, $a0 has value 0xff00
            - We expect that the new value in register $t1 will be 0xff0f.
        - and $t1, $t0, $a0 - $t1 has value 0x0137, $t0 has value 0xf00f, $a0 has value 0xff00
            - We expect the new value in register $t1 to be 0xf000
- ALU Output Register:
    - Description: This register is for the purpose of storing the value that was just calculated by the ALU.
    - Input: It is the 16-bit value that the ALU just calculated.
    - Output: It will have an output of a 16-value which will be used in various instructions.
    - Tests: We will do a branch and make sure that the value in ALUOut is the correct amount.
    - Input is 0x0005, so we expect that 0x0005 will be stored in ALUOutput register and that the output is also 0x0005.
        - beq $t0, $t0, 0x0003
            - We expect the program counter to increase by 4. (once at the very beginning and three more when the branch is run.

- PC Value Mux:
    - Description: This is the mux which controls how the PC Counter is updated
    - Input: It has six inputs that can be chosen. The value in Register A, which is a 16-bit value, the value directly after the ALU and before storing that value in ALUOut (16-bit value), and the ALUOut value which also has a 16-bit value. The top four bits of PC and bottom 12 bits of the instruction, a constant that represents the interrupt handling code address, the value from EPC.
    - Output: It will have an output of a 16-value which will go to the PC Register.
    - Control Signal: It is given a control signal to dictate how the PC Register will be updated and this is in the PC Value control.
    - Tests: We need to check when PC value is 0, 1, and 2. We will do a jump register and make sure it is set to 0. We will do an addition instruction and watch to make sure that the PC counter increases by 1 and that the PC value is a 1. We will also do a branch to test and make sure the PC value is a 2.
    - Coming in is 0x0001 in 0 and 0x0002 in 1 and 0x0003 in 2. If the mux is set to 0, we expect the output to be 0x0001. If the mux is set to 1, we expect the output to be 0x0002. If the mux is set to 2, we expect the output to be 0x0003.
        - jal $t0- $t0 has value 0x004 and we are at 0x0001
            - we expect the value in PC register to change to 0x0004 and the value of the line after 0x0001 : 0x0002 will be in the $ra register.
        - add $t1, $t0, $t0 - $t0 has value 0x000a and $t1 has value 0x0000
            - We expect the new value 0x014 to be in stored in register $t1
        - beq $t0, $t0, 0x0003
            - We expect the program counter to increase by 4 (once at the very beginning and three more when the branch is run).

- I or D mux:
    - Description: Instruction or Data mux to decide whether we will look up an instruction or whether we are going to do a store word.
    - Input: It has three inputs that can be chosen. The PC counter's value with a 16-bit value or the ALUOut value which also has a 16-bit value or a 16-bit value from a decoder.
    - Output: It will have an output of a 16-value which will go to the address in Memory.
    - Control Signal: It is given a control signal to dictate whether is will use the ALUOut value or the value of the PC Counter.
    - Tests: We will make sure a jump register will work and that the mux is set to a 0 to read the instruction. We will also do a store word to make sure that it is set to a 0 when loading the instruction and set to a 1 when storing the data in memory.
    - Coming in: 0x0001 in 0 and 0xffff and in 1. If the mux is 0, we expect the output to be 0x0001 and if the mux is 1 we expect the output to be 0xffff.
        - jal $t0 - $t0 has value 0x0004 and we are at 0x0001
            - we expect the value in PC register to change to 0x0004 and the value of the line after 0x0001 : 0x0002 will be in the $ra register.
        - sw $t0, ($a0)0 - $a0 has a value of 2 and $t0 is at memory 0x002a
            - We expect that 2 will be stored at the location 0x002a
- EPC register:
    - Description: Register to hold the program counter value when an interrupt occurs.
    - Input: One 16-bit input
    - Output: One 16-bit output
    - Control Signal: 1-bit EPCWrite
    - Tests: For any input value coming in when EPCWrite is 1, we expect that value out.
- Interrupt Register:
    - Description: Hold the values for the hardware that is triggering the interrupt. These values are used for determining what interrupt handler to run in response to the interrupt.
    - Input: Eight 1-bit values
    - Output: One 8-bit value
    - Control Signal: 4-bit value that will determine a bit to zero out.
    - Test: Initialize with some value, then send 4-bit values as control signals and see if the corresponding bits zero out.
- Decoder:
    - Description: Based on what interrupt is happening, certain values are selected and then sent as a possible input to memory.
    - Control signal: 4-bit selector

- Output: 16-bit address
- Tests: For each possible selector, one distinct address should be outputted.

Component Details

- PC Register
    - 16-bit register that will hold PC counter.
- Memory Block:
    - Block memory with a 16-bit input address to read and output the 16-bit value from, and a 16-bit data to write to the address if the MemWrite control signal is on. This will be initialized with a .coe file containing our program.
- Instruction Register:
    - 16-bit register that will hold the output of the Memory Block, and split that value into multiple outputs: a 4-bit OpCode (Instruction[15:12]), 4-bit Register Read Address (Instruction[11:8]), 4-bit Register Read Address (Instruction[7:4]), 8-bit Instruction[7:0]
- Memory Data Register:
    - 16-bit register that will also hold the output of the Memory Block.
- Sign-Extension (12 to 16 bit):
    - Sign-extend a 12-bit value to 16-bits.
- Register Write Address Mux:
    - A 4-input 4-bit Mux that will output a 16-bit value. These inputs will be Instruction[3:0], Instruction[7:4], Instruction[11:8], constant 0xb.
- Register Write Data Mux:
    - A 3-input 16-bit Mux that will output a 4-bit value. These inputs will be the extended Instruction[7:0], the Memory Data Register, and ALUOut.
- Sign-Extension (8 to 16 bit):
    - Sign-extend 8-bit to 16-bit.
- Sign-Extension (4 to 16 bit):
    - Sign-extend 4-bit to 16-bit.
- Load-Upper Extension (8 to 16 bit):
    - Create a 16-bit value as such: in: 0x12, out:0x1200. This can be done by taking the input and outputing input(7),input(6),input(5),input(4)input(3),input(2),input(1),input(0),gnd,gnd,gnd,gnd,gnd,gnd,gnd,gnd.
- Sign-Extended Value Mux:
    - This will take in two 16-bit values (the sign-extended 4-bit and 8-bit), and choose which will be used.
- General Register File:
    - A 16, 16-bit register file with 2, 4-bit read address inputs, a 4-bit write address, a 16-bit write data, 2, 16-bit read address outputs. The RegWrite control signal will write the write data to the write address.

- A Register:

    - 16-bit register that will hold the read address 1 output from the general register file. This is meant to be an input to the ALU Source A Mux for using register values with the ALU.
- B Register:
    - Description: Register to store the Read Out 2 value from General Reg. File. It has a 16 bit output value that goes into a mux to decide whether it will be used in the ALU.
- ALU Source A Mux:
    - Description: The unit for deciding what will go into the first half of the ALU. It is given two 16 bit value inputs. It then outputs the particular 16 bit value that is chosen.
- ALU Source B Mux:
    - Description: The unit for deciding what will go into the second half of the ALU. It is given four 16 bit value inputs. It then outputs the particular 16 bit value that is chosen.
- ALU:
    - Description: The unit for calculation within the processor. This takes care of all the arithmetic computations. It takes 2 16 bit value inputs and it will do some calculation decided upon by the ALU Operation through the bit code and then output the 16 bit value result.
- ALU Operation:
    - Description: This is the operation that we tell the ALU to use. We give it the op code as a 4 bit input and it decides what operation the ALU will use on its two inputs.
- ALU Output Register:
    - Description: This register is for the purpose of storing the value that was just calculated by the ALU. It is a 16 bit register that takes a 16 bit input that comes directly from the ALU. It has a 16 bit output that is the value that was stored in itself.
- PC Value Mux:
    - Description: This is the mux which controls how the PC Counter is updated. It is given three 16 bit values and will pick which one will update the PC counter.
- I or D mux:
    - Description: Instruction or Data mux to decide whether we will look up and instruction or whether we are going to do a store word. It takes two 16 bit values and will choose which will write to the address in memory.
- Main memory:
    - Description: Instruction memory and data memory. Can hold 2^14 instructions that can be read from and written to. Takes a 16 bit address but will only use the 14 least significant bits. DataIn is 16 bits and serves as write data. WriteEnabled controls writing to the register.

- Hardware Selector:

- Description: Determines which Hardware was just triggered

Control Signals

- IorD: This will choose between the PC, ALUOut, and decodeOut in a Mux whose output goes in as the address in the memory block.
- MemDataSelect: This will choose between ALUOut and lcdData in a Mux whose output goes in as the data in the memory block.
- lcdWrite: Writes to the lcdDisplay register.
- MemWrite: Write the input data to the input address in the memory block.
- IRegWrite: Write the input data to the Instruction Register.
- WriteAddr: Choose the general register address from a 4-input mux to write to.
- WriteData: Choose the data from a 4-input mux to write to the general register.
- RegWrite: Write the input data to the input write address in the general register file.
- ALUSrcA: Choose between PC and Register A from a 2-input mux as input for the ALU.
- ALUSrcB: Choose between Register B, 0x0001, a sign-extended value, and EPC from a 4-input mux as input for the ALU.
- PCData: Choose between Register A, the ALU output, ALUOut, alu result from a 5-input mux to write to the PC.
- SignExt: Choose between two sign-extended values to input to the ALUSrcB mux.
- PCWrite: Write the input to the PC register.
- PCWriteConditional: Write the input to the PC register (set when we're using conditional instructions such as branch equal).
- CLR: Chooses the bit in the interrupt register to clear. 4-bit signal.

Integration Testing

Our plan for integration testing is to work through the RTL specifications. Specifically, we will begin with the first block: PC = PC + 1, IR = MEM[PC], with tests to update the PC register, and store the value from memory location PC into the instruction register. Then, we'll test the next block, and we'll continue testing by going left-to-right through the RTL diagram (starting with A-Type instructions).

Update the PC reg

- PC starts at 0x0002
- ALUSrcA = 0
- ALUSrcB = 1
- PCData = 1
- Expect PC register to be 0x0003

Jump Register

- PC Starts at 0x0002
- Register A has value 0x0005
- IorD = 0
- PCData = 0
- Expect PC register to be 0x0005

Add $t0, $t1, $t2 -- $t0 has 0x0005, $t1 has value 0x000F, $t2 has value 0x000D

- IorD = 0
- ALUSrcA = 1
- ALUSrcB = 0
- GRegWrite = 1
- WriteData = 2
- WriteAddr = 2
- ALUOp = 2
- Expect register $t0 to have value 0x001C

Sub $t0, $t1, $t2 -- $t0 has 0x0005, $t1 has value 0x000F, $t2 has value 0x000D

- IorD = 0
- ALUSrcA = 1
- ALUSrcB = 0
- GRegWrite = 1
- WriteData = 2
- WriteAddr = 2
- ALUOp = 6
- Expect register $t0 to have value 0x0002

Beq $t0, $t0 , 0x0003 -- $t0 has value 0x0002 - PC register has value 0x0020

- PCWriteBeq = 1
- PCWrite = 0

- PCData = 2
- Expect PC to have value 0x0024

Control Unit

We drew out the data path and examined each instruction carefully. We figured out which values each mux would need to be set to in order to correctly execute the instruction. We listed out each portion of each instruction. We integrated our control designs using a Verilog State Machine. Our Control has 5 cycles. If an instruction takes less than 5 cycles, control can start the next instruction immediately.