The Eggo Stack

By:The Toaster Troop

William Dalby, Christian Meinzen, Victoria Szalay

Table of Contents

Synopsis:

The Eggo Stack (ES) architecture is a predominantly stack architecture created by the group known as the Toaster Troop. Said group consists of Victoria “Tori” Szalay, Christian Meinzen, and William “Will” Dalby. It operates with 16 bit words and has a 64 register deep stack. It has 16 operations which can be seen in the *List of Commands* section (pg 4). The semantics of how the stack works with these operations can be found in Appendix C (pg 32). This architecture has three instruction set designs consisting of A, B, and C type. These instruction sets have 4 bit opcodes. A-type handles arithmetic operations involving the stack, B-type handles branching and jumping operations, and C-type handles pushing and popping with immediates. Due to the fact that we are working with 16-bit words and 16-bit instructions, loading addresses into the stack is done in two parts. ES loads the upper byte of the immediate, then the lower (or vice versa) and or’s them together to put the full address at the top of the stack. Push and pop operations then look at this address at the top of the stack to find where the to push or pop from. A similar operation is performed for the jump stack (js) command which takes an address at the top of the stack and sets the PC equal to it. There are 4 reserved registers to hold a stack pointer, a global pointer, a zero value, and a return value. ES uses Pseudo Direct addressing for jumping and Base + Offset for branching. There is a 2^16 bit memory stack.0x0000 to 0x2000 is reserved for the kernel, 0x2000 to 0x4000 is for text, 0x4000 to 0x6000 is for Static Data, and from 0x6000 to 0x7ffe is for the Dynamic Data which grows down.

Patch Notes:

- Methodology for pushing things to the memory stack has changed. Instead of pushing the upper and lower immediates of the memory address we wish to store in, it is relative to a Stack Pointer. The callee must move their stack pointer and then move it back upon return.
- Added a funct to the C type instruction to allow for pushing to memory and registers separately.
- Changed the A-type ISA to have a duplicate amount along with a shamt.
- The procedure calling was changed in that instead of putting in an address to return to, we push a label that is after the jump.
- Updated ISA, common operations, and relprime code to reflect these changes
- Some of the inputs for the hardware have been changed to be control bits. For example, flip is now a control bit on the execution stack rather than an input.
- The memory addresses can only be 10 bits long so the input to memory to read or write to has been clipped to 10 bits.
- Shift methodology has been changed. We now send a signed shamt to the shifter and it shifts the appropriate amount in the appropriate direction (left > 0, right < 0).
- The state transition diagram was updated to include all instructions.
- The datapath was changed to make the input to the shift component 16 bits with the sign extender since our original implementation invalidly inputted an 8 bit value.

List of Commands:

Command	Opcode	Funct	Type
pushM	0x0 / 0b0000	0b00	C
popM	0x1 / 0b0001	0b00	C
pushR	0x0 / 0b0000	0b01	C
popR	0x1 / 0b0001	0b01	C
pushli	0x2 / 0b0010	n/a	C
pushui	0x3 / 0b0011	n/a	C
dup	0x4 / 0b0100	n/a	A
flip	0x5 / 0b0101	n/a	A
or	0x6 / 0b0110	n/a	A
add	0x7 / 0b0111	n/a	A
sub	0x8 / 0b1000	n/a	A
lsl	0x9 / 0b1001	n/a	A
lsr	0xA / 0b1010	n/a	A
slt	0xB / 0b1011	n/a	A
beq	0xC / 0b1100	n/a	B
bne	0xD / 0b1101	n/a	B
j	0xE / 0b1110	n/a	B
js	0xF / 0b1111	n/a	B

Reserved Registers:

1. Stack Pointer - SP = 0b00
2. Return Value - V0 = 0b01
3. Global Pointer - GP = 0b10
4. Zero Register - zero = 0b11

ISA designs:

Arithmetic Items: A-Type

4 bits	6 bits	4 bits	2 bit
OPCODE	UNUSED	SHAMT	DAMT (Duplicate amount)

Jumping / Branching: B-Type

4 bits	12 bits
OPCODE	ADDRESS/IMMEDIATE

Items involving immediates: C-type

4 bits	8 bits	2 bit	2 bits
OPCODE	IMMEDIATE	FUNCT	RESERVED REGISTER

Addressing Modes:

Pseudo Direct: For jumping - 12 bits of address and 4 bits from PC
Base + Offset: For branching - Number of instructions from PC+2

Procedure Calling Convention:

The return address is placed at the top of the stack. Arguments are stored at the top of the stack. The Procedure is then called. The return value is stored in Mem[V0] by the callee. Caller then retrieves this and puts it on the stack. This means that when the caller regains control of the stack, it will contain only old values.

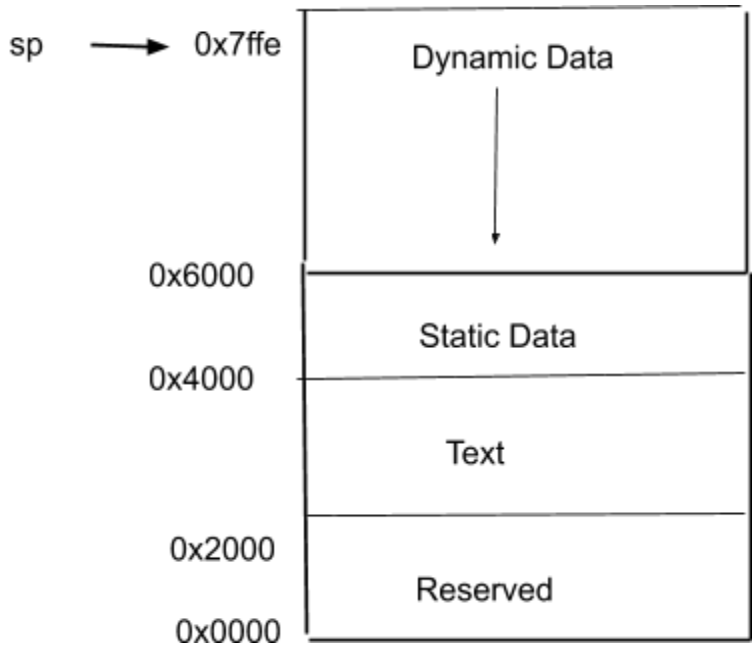
Local Variable Convention:

The programmer is responsible for storing local variables in the memory stack. This is done using push/pop M in reference to the stack pointer.

Chosen Size of Register Stack:

We have chosen the size of our register stack to be constructed out of 64 16-bit available registers. This decision was made because this is the maximum size that the FPGA board allows.

Memory Allocation:



RTL:

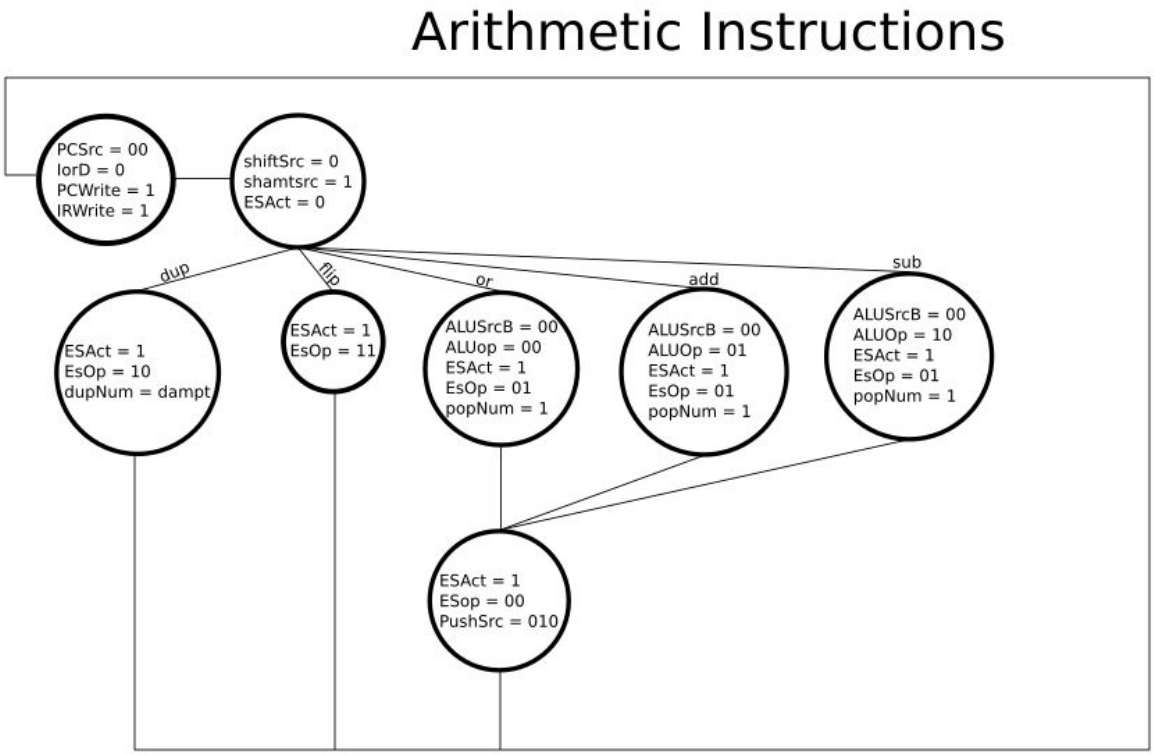
Step	Push/Pop M	Push/Pop R	Arithmetic/ Logic	beq/bne	j	js
Inst Fetch	newPC = PC + 2 PC = newPC Inst = Mem[PC]					
Inst Decode Pop from stack pushLI/ UI done dupAMT fetched	A = ES[Top] B = ES[Top + 2]					
Execution Beq / j / js done Address Comp PopM done push/Pop R done Flip done Shift done Duplication occurs depending on dupAMT	ALUout = A + SE(inst[11-4]) pushM: ES(pop 0) Memout = Mem[ALUout] popM: ES(pop 1) Mem[ALUout] = B	popR: ES(pop 0) Reg[inst[1-0]] = A pushR: ES(push reg[inst[1-0]])	ALUout = A op B ES(pop 1)	ES(pop 1) If (A==B) then PC =PC[15-11] || inst[11-0]	PC = PC[15-11] || inst[11-0]	ES(pop 0) Address = A
Push to stack Arithmetic / Logic Done pushM done	pushM: ES(push Memout)		ES(Push ALUout)			PC = address

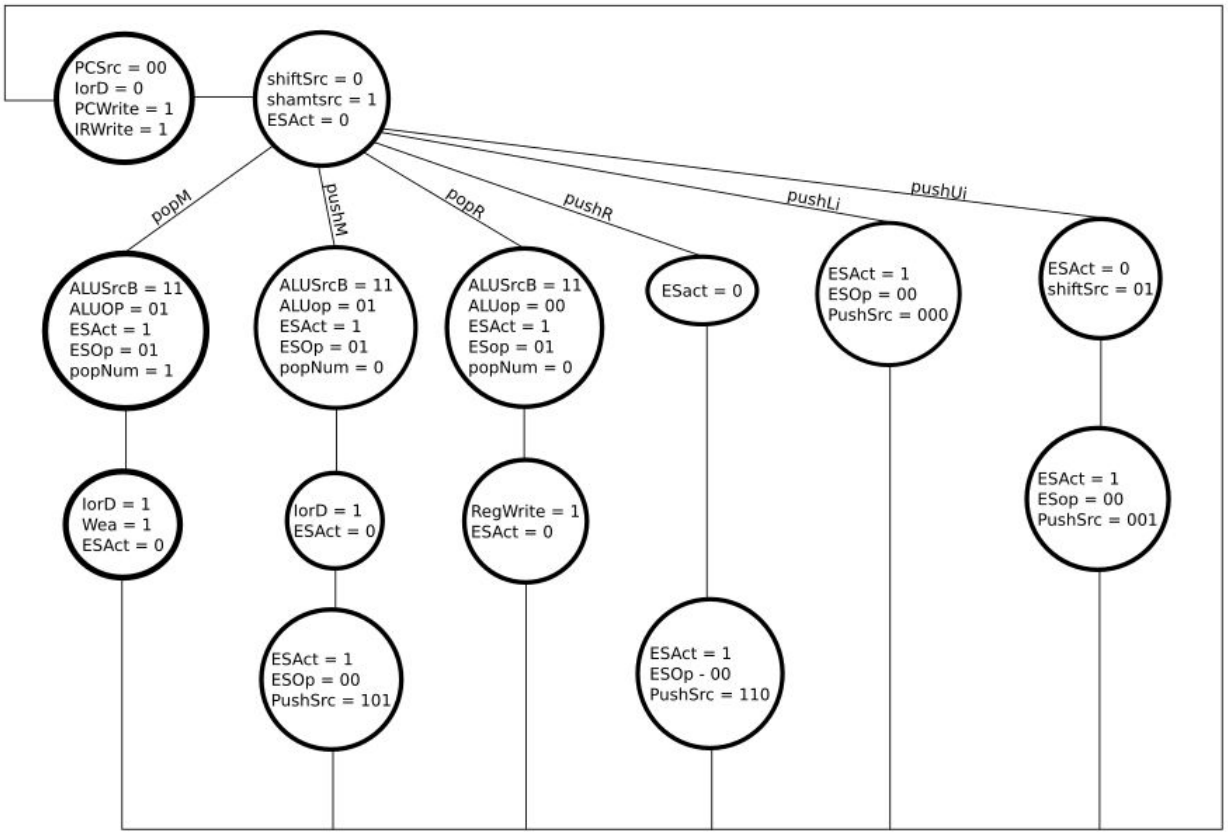
Step	slt	lsr/ lsl	flip	dup	pushUI/LI
Inst Fetch	newPC = PC + 2 PC = newPC Inst = Mem[PC]				
Inst Decode Pop from stack pushLI/ UI done dupAMT fetched	A = ES[Top] B = ES[Top + 2]				
Execution Beq / j / js done Address Comp PopM done push/Pop R done Flip done Shift done Duplication occurs depending on dupAMT	ES(pop 1) ALUout = A<B	A = A shifted by inst[5-2] (shamt) in the appropriate direction	ES[top] = ES[top+1] ES[top+1] = ES[top]	dupAMT = inst[1-0] If dupAMT == 0: ES[Top+2] = ES[Top] Top = Top+2 Done If dupAMT == 1: ES[Top+2] = ES[Top-2] ES[Top+4] = ES[Top] Top = Top+4 If dupAMT==2: ES[Top+2] = ES[Top-4] ES[Top+4] = ES[Top -2] ES[Top+6] = ES[Top] Top = Top + 6 Else: ES[Top+2] = ES[Top - 6] ES[Top+4] = ES[Top-4] ES[Top+6] = ES[Top-2] ES[Top+8] = ES[Top] Top = Top + 8	UI: ES(push SE(inst[11-4]) << 2) LI:ES(push ZE(inst[11-4]))

Push to stack
Arithmetic / Logic Done
	IF(ALUout = 1)
	ES(push 1)
	
	ELSE
	ES(push 0)
		Push A			

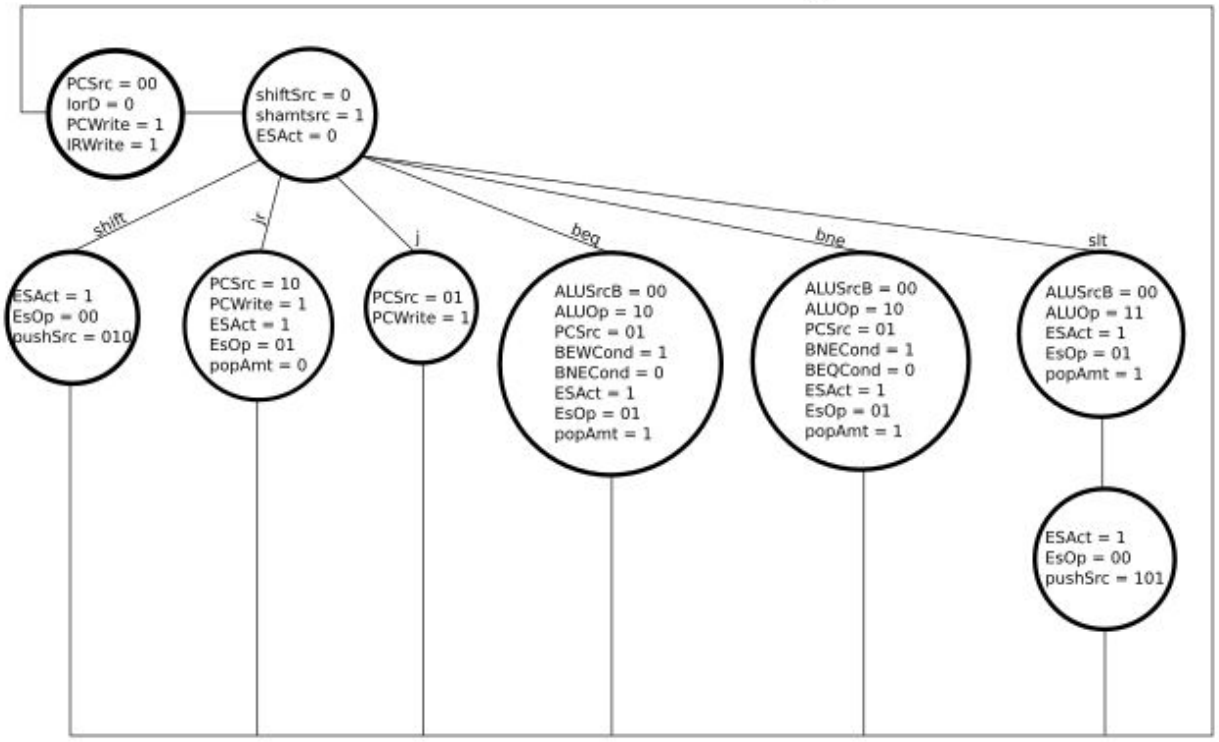
pushM done					

State Transition Diagram for Multicycle:

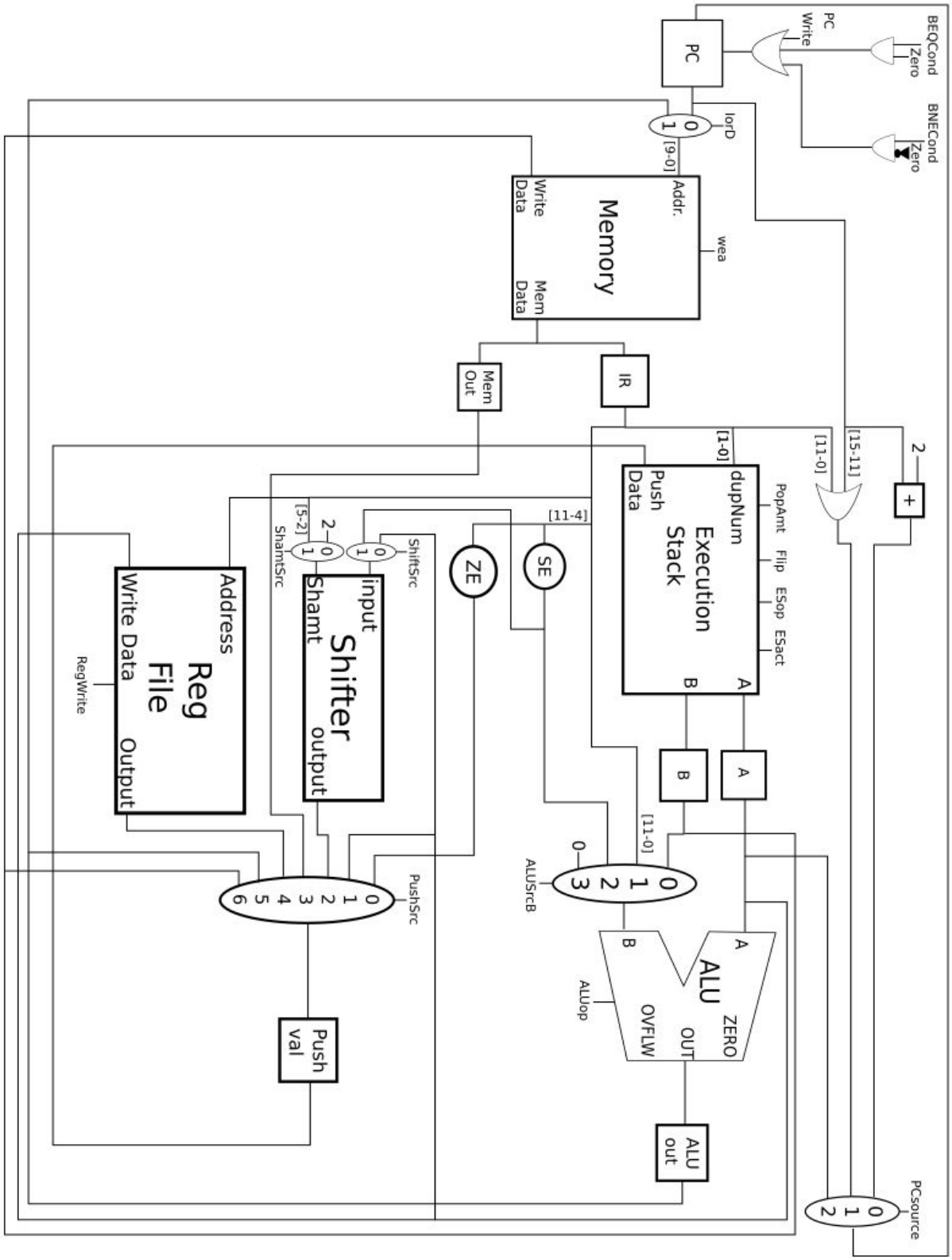




SLT and Shifting



Datapath:


BEQCond
Zero
PC Write
PC
BNECond
Zero
IorD
0
1
[9-0]
Addr.
Write Data
Memory
Mem Data
wea
Mem Out
IR
[1-0]
dupNum
Push Data
Execution Stack
PopAmt
Flip
ESop
ESact
B
A
[11-0]
[15-11]
2
+
[11-4]
ZE
SE
[5-2]
ShamtSrc
ShiftSrc
input
Shamt
Shifter
output
Address
Reg File
Write Data
Output
RegWrite
B
A
ALUSrcB
ALUop
ALU
ZERO
OVFLW
OUT
ALU out
PushSrc
Push val
PCsource

Control Symbol Tables:

PCSrc	
Value	**Source**
00	PC + 2
01	PC[15-11] | IR[11-0]
10	A

ALUSrcB	
Value	**Source**
00	B
01	Inst[11-0]
10	SE(inst[11-0])
11	0

ALUOp	
Value	**Operation**
00	Or
01	Add
10	Sub
11	SLT

ESOp	
Value	**Operation**
00	push
01	pop (popNum)
10	dup (dupNum)
11	flip

PushSRC	
Value	Source
000	ZE(inst[11-4])
001	A
010	ShiftOut
011	MemOut
100	RegFile
101	ALUOut
110	B

ShamtSrc	
Value	Source
0	2
1	inst[5-2]

ShiftSrc	
Value	Source
0	A
1	inst[11-4]

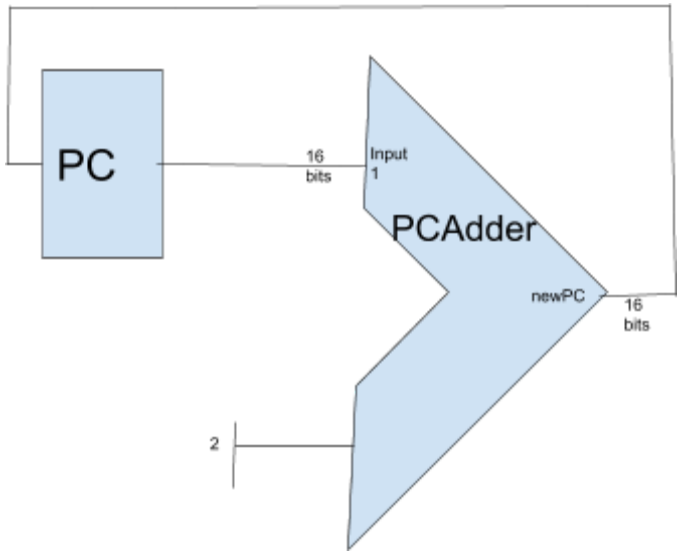
IorD	
Value	Source
0	PC
1	inst[11-4]

RTL Verification:

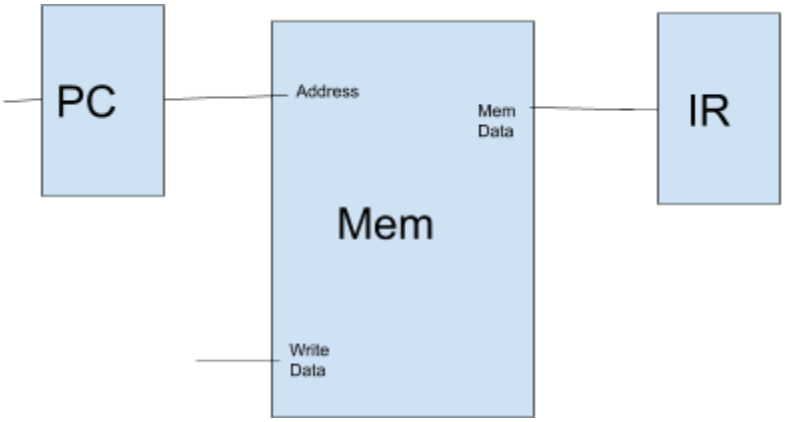
We verified our register transfer language by simulating the implementation of each instruction through the described above components. Below, we will show how we simulated add:

Add

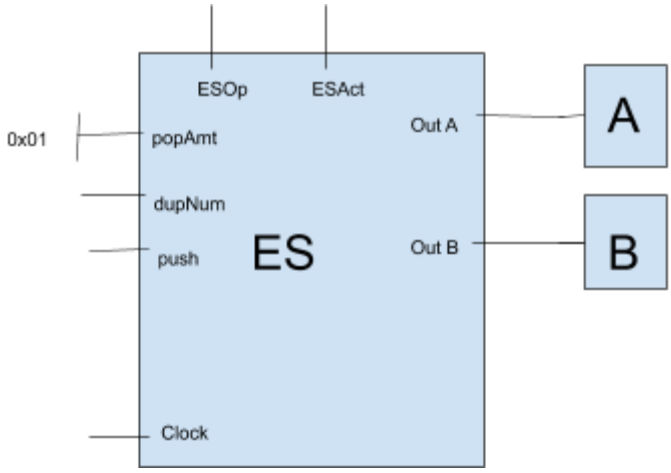
1. ***newPC = PC + 2***
2. ***PC = newPC***



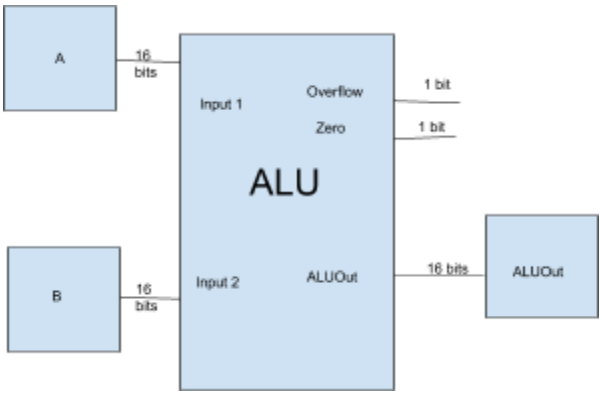
3. ***Inst = Mem[PC]***



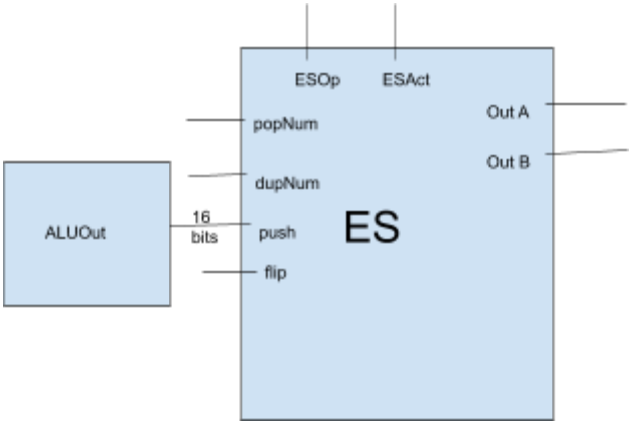
4. ***A = pop***
5. ***B = pop***



6. ALUout = A+B



7. Push ALUout



Testing Methodology:

We used to several different testing methodologies while testing the processor.

1. Component Testing:
    a. For each component test, we decided to test all functions provided by the component and any edge cases presented by those functions.
2. Integration Testing:
    a. We integrated our components together by sections of the overall datapath that will perform operations together. After combining them, we hard-coded inputs at the beginning of the section and checked the outputs to see that they were what we expected.
3. System Testing:
    a. We plan to first completely test the datapath without the control unit attached and once the datapath successfully completes all types of instructions with inputted control signals, we will attach the control unit. We will then test the control unit with a single instruction, and then followed with a series of instructions.

Integration Plan:

1. Subsystem level 1
    a. PC and PCAdder **--completed**
        i. **Testing plan:** Be able to increment PC's value by 2 using PCAdder
    b. Execution Stack, AB registers, ALU -- **completed**
        i. **Testing plan:** Be able to store values from execution stack into A,B registers and have those registers input into the ALU.
            1. Put manual inputs into execution stack, clock the stack, clock the registers to go into the ALU, clock the stack for the result to be inputed, check that the stack has changed
            2. *Manual inputs*:
                a. Test Add: push 1 and 2 onto the stack; wait for the clock to go through datapath; check to see that 3 is in register A and on the stack
                b. Test Or: Same as before except should be 0 after.
                c. Test Sub: Same as before but should be 1 after.
                d. Test lessThan: Same as before but should be 0 after.
    c. Memory, IR, SE, ZE,Shifter -**completed**
        i. **Testing plan:** Be able to retrieve instructions from .coe and store it into IR register. Have IR register's output feed into SE, ZE, and Shifter. Check that actual values match expected ones. These edge cases were tested
            1. Check that when IR[11-4] has a leading 0 that the correct value is outputted in both SE and ZE
            2. Check that when IR[11-4] has a leading 1 that the correct value is outputted in both SE and ZE

3. Check that when IR[5-2] is positive that shifter shifts to the left and that it outputs the correct value
4. Check that when IR[5-2] is negative that shifter shifts to the right and it outputs the correct value

d. PushSrcMux, all Push Sources, and Execution Stack **-- completed**
   i. **Testing plan:** Be able to input into all 7 possible push sources, manually set the mux, and make sure execution stack properly pushes value onto the stack
      1. Put manual inputs into pushVal mux and execution stack, clock the registers to see that the stack is changed.
      2. Make sure that using different functions on the execution stack are not affected by different push sources
      3. *Manual Inputs:*
         a. Test 1: Put [0,1,2,3,4,5] into the first six mux inputs and the A and B registers from the output of ES into the sixth and seventh input of the mux. Check that each input changes the stack properly by assigning PushSrc = 0, then PushSrc = 1, … and so on. ESOp = 0 and ESAct = 1 for this test.
         b. Test 2: Make ESOp = 1 and check that nothing happens to the stack except that tos is updated as well as registers A and B. Do the same with ESAct = 0.

e. ALU, ALUOut, Memory,Memout **--completed**
   i. **Testing plan:** Be able to feed ALU's results to memory and store a value to the correct calculated address.Hardcode ALU's inputs as well as memory's write data. Based on wea's value, check that the correct address has either been read or written to with the correct value. This path should only be for the popM and pushM instructions and the ALU always adds.
      1. To check that calculated addresses can be read:
         a. Set wea = 0
         b. Set ALUOp = 01
         c. Set ALUOut write enable = 1
         d. Set memory's write data = 0
         e. ALUinput1 = 0x0000
         f. ALUinput2 = 0x0000
         g. Check that address 0x0000's value that is initialized into the .coe file is correctly stored into register Memout.
      2. To check that calculated addresses can be written to:
         a. Set wea = 1
         b. Set ALUOp = 01
         c. Set ALUOut write enable = 1
         d. Set memory's write data = 0x8888
         e. ALUinput1 = 0x1232

f. ALUinput2 = 0x0002
g. Check that address 0x1234's value of 0x8888 is correctly stored into register Memout
3. Repeat for several values.

f. Memory, IR, Reg File, Execution Stack **-- completed**
   i. **Testing plan:** Be able to store a value from memory to the execution stack, put that same value into a register file, push it back onto the stack, and pop it off to see if the system worked properly. Need to use a push value mux.
      1. Check that data is pushed from memory onto the stack:
         a. *Manual Inputs:* pushSrc = 0, ESOp = 0, ESAct = 1. CLK, ESAct=0, CLK. wea = 0, pushVal = 2, CLK.
      2. Check that the data gets stored in register file:
         a. Manual Inputs: ESAct = 0, CLK. ESAct = 1, ESOp = 1, popNum = 0, regWrite = 1, regAddress = 0, CLK.
      3. Check that you can write to memory:
         a. Manual Inputs: ESAct = 1; push_in = 15; pushSrc = 1; ESOp = 0; CLK. pushSrc = 0; CLK. ESAct = 0; CLK. wea = 1; ESAct = 1; popNum = 1; ESOp = 1; CLK.

g. Memory, Control **--completed**
   i. **Testing plan:** Be able to read a series of machine code instructions from the.coe file and have the control set the appropriate bits for all the states
      1. .coe file is filled with machine code for all 18 instructions
      2. Have the testbench initial block read all eighteen instructions
         a. An instruction is read every five clock cycles
      3. Check that all states are appropriately handled for that instruction

2. Create datapath based off of corrections and considerations obtained from sublevel 1

System Testing Plan:

1. Complete a single instruction through datapath with manually set controls.
   a. A type instruction testing
      i. We will set all control values to perform one instance of the instruction Add. We will preload values into each of the necessary modules to do so.
         1. PC will be set to an address which contains an add instruction
         2. Memory will have an address with an add instruction
         3. The Execution Stack will contain the 2 values that we wish to add
         4. All control bits will be set manually
            a. See multicycle diagrams
         5. We will then check the output of the ALU and the Execution Stack to see if the sum was pushed back onto the Stack
   b. B type instruction Testing
      i. We will set all control values to perform one instance of the instruction j

1. PC will be set to an address which contains a j instruction
2. Memory will have an address with a j instruction
3. All control values will be set manually
    a. See multicycle diagrams
4. We will then see if the PC has the correct value that we wished to Jump to

c. C type instruction testing
    i. We will set all control values to perform one instance of the instruction pushR
        1. PC will be set to an address which contains a pushR instruction
        2. Memory will have an address with a pushR instruction
        3. The register file will have a register with a known value (5) to read
        4. All control values will be set manually
            a. See multicycle diagrams
        5. We will then check the execution stack to see if the value from the register file is in the stack

2. Complete a single instruction through datapath with connected control unit.
    a. All processes, instructions, and result checking will be the same as in the manual portion but now the control unit will set the control bits.
3. Complete a series of instructions with connected control unit

Which series?

Appendix A: RelPrime Code

Address/Label	**Assembly**	**Machine**	**Comments**
0x2000	pushLI 2	0b 0010 0000 0010 0000	
0x2002	pushR $sp	0b 0000 0000 0000 0100	Pop M
0x2004	popM 0	0b 0001 0000 0000 0000	From stack
0x2006	pushR $sp	0b 0000 0000 0000 0100	Pop N
0x2008	popM -2	0b 0001 1111 1110 0000	From stack
0x200A	pushLI 1	0b 0010 0000 0001 0000	
0x200C / LOOP	pushLI RETURN	0b 0010 0010 0110 0000	Push RETURN
0x200E	pushUI RETURN	0b 0011 0010 0000 0000	to stack
0x2010	pushR $sp	0b 0000 0000 0000 0100	Push M
0x2012	pushM 0	0b 0000 0000 0000 0000	To Stack
0x2014	pushR $sp	0b 0000 0000 0000 0100	Push N
0x2016	pushM -2	0b 0000 1111 1110 0000	To Stack
0x2018	pushLI 0xfa	0b 0010 1111 1010 0000	Sets up SP
0x201A	pushUI 0xff	0b 0011 1111 1111 0000	For use
0x201C	or	0b 0110 0000 0000 0000	In GCD
0x201E	pushR $sp	0b 0000 0000 0000 0100	By decrementing
0x2020	add	0b 0111 0000 0000 0000	By 6: the # of
0x2022	popR $sp	0b 0001 0000 0000 0100	Vars used here
0x2024	j GCD	0b 1110 0000 0100 1100	
0x2026 / RETURN	pushLI 0x06	0b 0010 0000 0110 0000	Destroys SP
0x2028	pushR $sp	0b 0000 0000 0000 0100	After coming
0x202A	add	0b 0111 0000 0000 0000	Back from
0x202C	popR $sp	0b 0001 0000 0000 0100	GCD
0x202E	pushR $v0	0b 0000 0000 0000 0101	Push V0

0x2030	pushM -4	0b 0000 1111 1100 0000	To Stack
0x2032	bne DONE1	0b 1101 0000 0100 0010	
0x2034	pushR $sp	0b 0000 0000 0000 0100	Push M
0x2036	pushM 0	0b 0000 0000 0000 0000	To stack
0x2038	pushLI 1	0b 0010 0000 0001 0000	
0x203A	add	0b 0111 0000 0000 0000	
0x203C	pushR $sp	0b 0000 0000 0000 0100	Pop M
0x203E	popM 0	0b 0001 0000 0000 0000	To Stack
0x2040	j LOOP	0b 1110 0000 0000 1100	
0x2042 / DONE1	pushR $sp	0b 0000 0000 0000 0100	Push M
0x2044	pushM 0	0b 0000 0000 0000 0000	To stack
0x2046	pushR $v0	0b 0000 0000 0000 0101	Pop V0
0x2048	popM -4	0b 0001 1111 1100 0000	From Stack
0x204A[1]	js	0b 1111 0000 0000 0000	
0x204C/ GCD[2]	pushR $sp	0b 0000 0000 0000 0100	B is on the stack
0x204E	popM 0	0b 0001 0000 0000 0000	(arg) pop to mem
0x2050	pushR $sp	0b 0000 0000 0000 0100	A is on the stack
0x2052	popM -2	0b 0001 1111 1110 0000	Pop it to mem
0x2054	pushR $sp	0b 0000 0000 0000 0100	
0x2056	pushM 0	0b 0000 0000 0000 0000	
0x2058	pushR $sp	0b 0000 0000 0000 0100	
0x205A	pushM -2	0b 0000 1111 1110 0000	
0x205C	bne LOOP2	0b 1101 0000 0110 0000	Knocks A and B
0x205E	js	0b 1111 0000 0000 0000	Off, jumps to RA

[1] This is the last instruction in the relPrime program
[2] GCD denotes the beginning of the GCD program

0x2060 / LOOP 2	pushR $sp	0b 0000 0000 0000 0100	
0x2062	pushM 0	0b 0000 0000 0000 0000	
0x2064	pushLI 0	0b 0010 0000 0000 0000	
0x2066	bne DONE2	0b 1101 0000 1001 1010	
0x2068	pushR $sp	0b 0000 0000 0000 0100	
0x206A	pushM 0	0b 0000 0000 0000 0000	
0x206C	pushR $sp	0b 0000 0000 0000 0100	
0x206E	pushM -2	0b 0000 0000 0000 0000	
0x2070	slt	0b 1011 0000 0000 0000	
0x2072	pushLI 0	0b 0010 0000 0000 0000	
0x2074	bne CON1	0b 1101 0000 0001 0010	
0x2076	pushR $sp	0b 0000 0000 0000 0100	
0x2078	pushM -2	0b 0000 0000 0000 0000	
0x207A	pushR $sp	0b 0000 0000 0000 0100	
0x207C	pushM 0	0b 0000 0000 0000 0000	
0x207E	sub	0b 1000 0000 0000 0000	
0x2080	pushR $sp	0b 0000 0000 0000 0100	
0x2082	pushM 0	0b 0000 0000 0000 0000	
0x2084	pushR $sp	0b 0000 0000 0000 0100	
0x2086	popM 0	0b 0001 0000 0000 0000	
0x2088	J LOOP2	0b 1110 0000 0110 0000	
0x208A / CON1	pushR $sp	0b 0000 0000 0000 0100	
0x208C	pushM 0	0b 0000 0000 0000 0000	
0x208E	pushR $sp	0b 0000 0000 0000 0100	
0x2090	pushM -2	0b 0000 1111 1110 0000	
0x2092	sub	0b 1000 0000 0000 0000	

0x2094	pushR $sp	0b 0000 0000 0000 0100	
0x2096	popM -2	0b 0001 1111 1110 0000	
0x2098	J LOOP2	0b 1110 0000 0110 0000	
0x209A / DONE2	pushR $sp	0b 0000 0000 0000 0100	
0x209C	pushM -2	0b 0000 1111 1110 0000	
0x209E	PushR $v0	0b 0000 0000 0000 0101	
0x20A0	pushM 0	0b 0000 0000 0000 0000	
0x20A2	js	0b 1111 0000 0000 0000	

Appendix B: Common Code Snippets

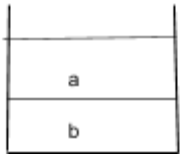
1. Conditional Statements:

a. Less than

C code:

```
if(a < b){
      a = a + b
}
else{
      b = a + b
}
```

ASSUMPTION;
Initial stack before if statement:



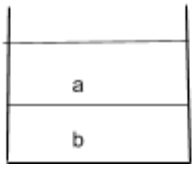
Address	Assembly	Machine Code
0x2000	PushR $sp	0b 0000 0000 0000 0100
0x2002	pushM 0	0b 0000 0000 0000 0000
0x2004	PushR $sp	0b 0000 0000 0000 0100
0x2006	pushM -2	0b 0000 1111 1110 0000
0x2008	add	0b 0111 0000 0000 0000
0x200A	PushR $sp	0b 0000 0000 0000 0100
0x200C	pushM 0	0b 0000 0000 0000 0000
0x200E	PushR $sp	0b 0000 0000 0000 0100
0x2010	pushM -2	0b 0000 1111 1110 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR $zero	0b 0000 0000 0000 0111
0x2016	Bne DOIT	0b 1101 0000 0011 0000
0x2018	PushR $sp	0b 0000 0000 0000 0100
0x201A	popM -2	0b 0001 1111 1110 0000
0x2030 / DOIT	PushR $sp	0b 0000 0000 0000 0100
0x2032	popM 0	0b 0001 0000 0000 0000

b. Greater than or equal

C code:

```
if(a >= b){
      a = a + b
}
else{
      b = a + b
}
```

ASSUMPTION;
Initial stack before if statement:



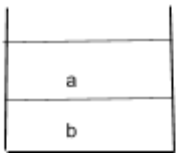
Address	**Assembly**	**Machine**
Items up to the slt are the same as in the less than code		
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR $zero	0b 0000 0000 0000 0111
0x2016	Beq DOIT	0b 1100 0000 0011 0000
Items in the DOIT section are the same as in the less than code		

c. Less than or equal

C code:

```
if(a <= b){
      a = a + b
}
else{
      b = a + b
}
```

ASSUMPTION;
Initial stack before if statement:



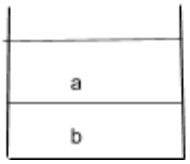
Address	**Assembly**	**Machine**
Items up to the slt are the same as in the less than code		
0x2010	flip	0b 0101 0000 0000 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR $zero	0b 0000 0000 0000 0111
0x2016	Beq DOIT	0b 1100 0000 0011 0000
Items in the DOIT section are the same as in the less than code		

d. Greater than

C code:

```
if(a > b){
      a = a + b
}
else{
      b = a + b
}
```

ASSUMPTION;
Initial stack before if statement:



Address	**Assembly**	**Machine**
Items up to the slt are the same as in the less than code		
0x2010	flip	0b 0101 0000 0000 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	pushR $zero	0b 0000 0000 0000 0111
0x2016	Bne DOIT	0b 1100 0000 0011 0000
Items in the DOIT section are the same as in the less than code		

2. Looping

a. While loop

C code:

```
a = 0;
b = 0;
while (a < 10){
      b = b + a;
      a++;
}
```

ASSUMPTION;
Initial stack looks like:

Address	Assembly	Machine Code
0x2000	pushR $zero	0b 0000 0000 0000 0111
0x2002	pushR $sp	0b 0000 0000 0000 0100
0x2004	dup 1	0b 0100 0000 0000 0001
0x2006	popM 0	0b 0001 0000 0000 0000
0x2008	popM -2	0b 0001 1111 1110 0000
0x200A / LOOP	pushR $zero	0b 0000 0000 0000 0111
0x200C	pushR $sp	0b 0000 0000 0000 0100
0x200E	pushM 0	0b 0000 0000 0000 0000
0x2010	pushLi 10	0b 0010 0000 1010 0000
0x2012	slt	0b 1011 0000 0000 0000
0x2014	beq END:	0b 1100 0000 0011 0000
0x2016	pushR $sp	0b 0000 0000 0000 0100
0x2018	pushM 0	0b 0000 0000 0000 0000
0x201A	dup 0	0b 0100 0000 0000 0000
0x201C	pushR $sp	0b 0000 0000 0000 0100
0x201E	pushM -2	0b 0000 1111 1110 0000
0x2020	add	0b 0111 0000 0000 0000
0x2022	pushR $sp	0b 0000 0000 0000 0100
0x2024	popM -2	0b 0001 1111 1110 0000
0x2026	pushLi 1	0b 0010 0000 0001 0000
0x2028	add	0b 0111 0000 0000 0000
0x202A	pushR $sp	0b 0000 0000 0000 0100
0x202C	popM 0	0b 0001 0000 0000 0000
0x202E	j LOOP	0b 1110 0000 0000 1010
0x2030 / DONE	OTHER CODE	

b. For loop

```
C code:
int b = 0;
for(int a = 0; a<10, a++){
      b = b + a;
}
```

ASSUMPTION;
Initial stack looks like:

The machine and assembly code is the same as the code for the while loop.

3. Procedure Calling

a. Caller to Callee

C code:
(*some code*)
a = getGCD(a,b);

ASSUMPTION;
Initial stack looks like:

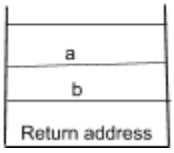
Address	Assembly	Machine
0x200C	pushLI RETURN	0b 0010 0010 0110 0000
0x200E	pushUI RETURN	0b 0011 0010 0000 0000
0x2010	pushR $sp	0b 0000 0000 0000 0100
0x2012	pushM 0	0b 0000 0000 0000 0000
0x2014	pushR $sp	0b 0000 0000 0000 0100
0x2016	pushM -2	0b 0000 1111 1110 0000
0x2018	pushLI 0xfa	0b 0010 1111 1010 0000
0x201A	pushUI 0xff	0b 0011 1111 1111 0000
0x201C	or	0b 0110 0000 0000 0000
0x201E	pushR $sp	0b 0000 0000 0000 0100
0x2020	add	0b 0111 0000 0000 0000
0x2022	popR $sp	0b 0001 0000 0000 0100
0x2024	j GCD	0b 1110 0000 0100 1100
0x2026 / RETURN	pushLI 0x06	0b 0010 0000 0110 0000
0x2028	pushR $sp	0b 0000 0000 0000 0100
0x202A	add	0b 0111 0000 0000 0000
0x202C	popR $sp	0b 0001 0000 0000 0100
0x202E	pushR $v0	0b 0000 0000 0000 0101
0x2030	pushM -4	0b 0000 1111 1100 0000
0x2032	bne DONE1	0b 1101 0000 0100 0010

b. Callee to Caller

C code:

```
int getGCD(int a, int b){
    ...
    ...
    return a;
}
```

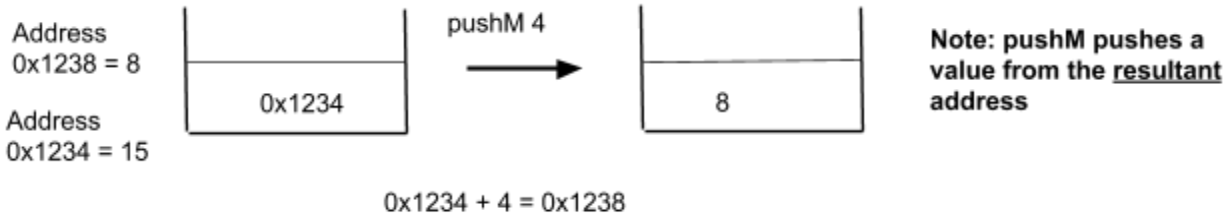
ASSUMPTION;
Initial stack looks like:



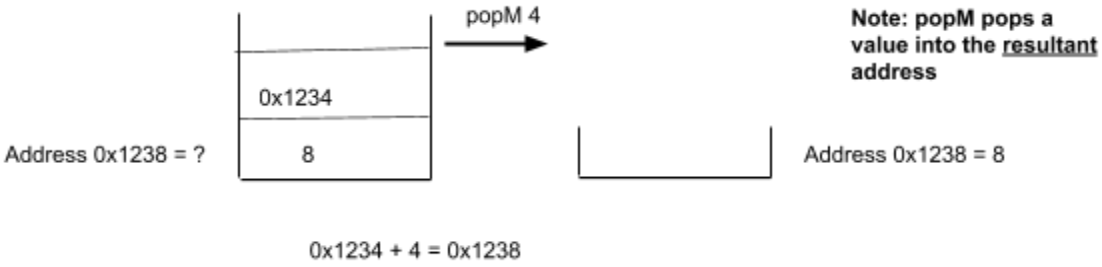
Address	**Assembly**	**Machine**
0x2040	pushR $sp	0b 0000 0000 0000 0100
0x2042	popM 0	0b 0001 0000 0000 0000
0x2044	pushR $sp	0b 0000 0000 0000 0100
0x2046	popM -2	0b 0001 1111 1110 0000
	OTHER CODE	
0x2060	pushR $sp	0b 0000 0000 0000 0100
0x2062	pushM 0	0b 0000 0000 0000 0000
0x2064	pushR $V0	0b 0000 0000 0000 0111
0x2066	popM 0	0b 0001 0000 0000 0000
0x2068	js	0b 1111 0000 0000 0000

Appendix C: Instruction Description and Semantics

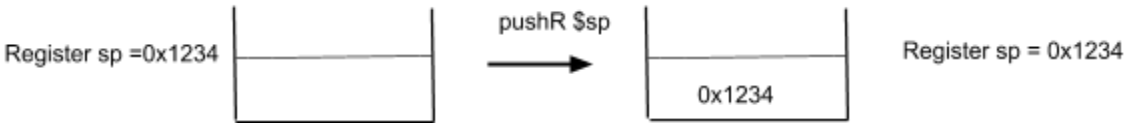
1. **pushM** takes the 16-bit address from the top of the stack, adds an immediate to that address, and returns the value that is stored in the altered address
   ISA: C-type
   Example: **pushM 2**
   Visualization of the stack



2. **popM** takes the 16-bit address that is on the top of the stack, adds an immediate to the address, and stores the value below that into the resultant address
   ISA: C-type
   Example: **popM 4**
   Visualization of the stack

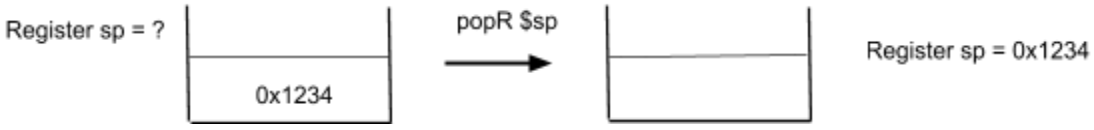


3. **pushR** pushes a 16 bit value from a specified register onto the stack.
   ISA: C-type
   Example: **pushR $sp**
   Visualization of the stack

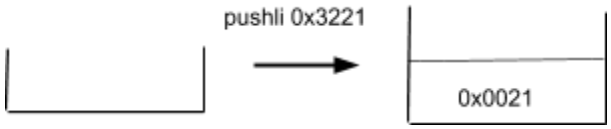


4. **popR** stores a 16 bit value from the top of the stack into the specified register. The value is then popped off the stack.
   ISA: C-type
   Example: **popR $sp**

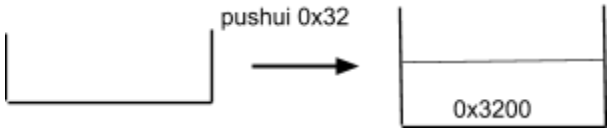
Visualization of the stack



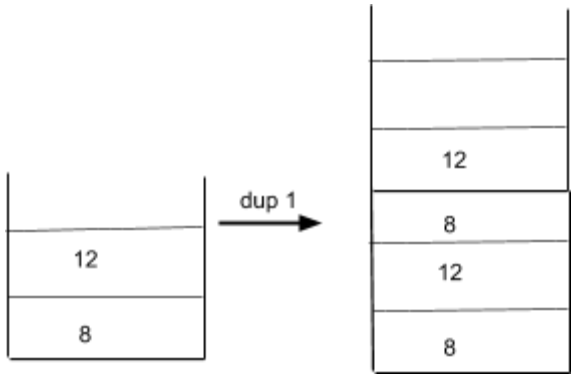
5. **pushli** takes the lower 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack
   ISA: C-type
   Example: **pushli 0x3221**
   Visualization of the stack



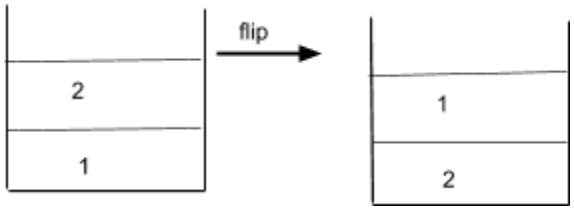
6. **pushui** takes the upper 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack.
   ISA: C-type
   Example: **pushui 0x32**
   Visualization of the stack



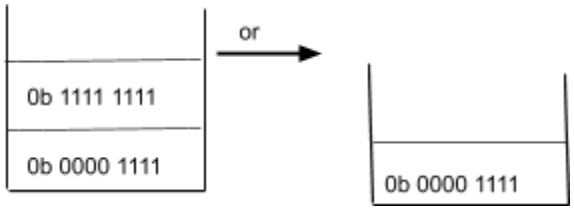
7. **dup** looks at the specified amount of data from the top of the stack, copies the data, and pushes it on to the top of the stack
   ISA: A-type
   Example: **dup 1**



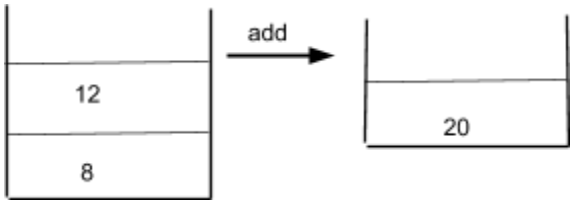
8. **flip** takes the two topmost values in the stacks and reverses their order on the stack.
   ISA: A-type
   Example: **flip**
   Visualization of stack



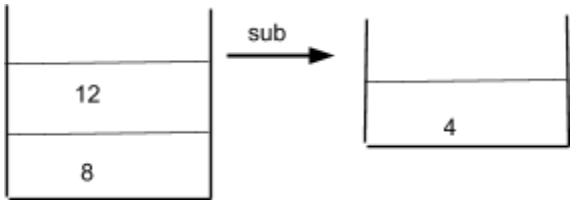
9. **or** looks at the two topmost values of the stack and performs the bitwise ‘or’ operation. The result is stored at the top of the stack.
   ISA: A-type
   Example: **or**
   Visualization of stack



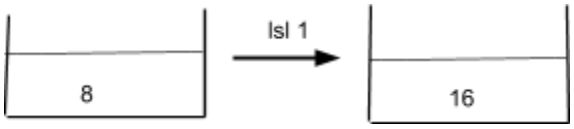
10. **add** takes the two values stored at the top of the stack, adds the values, and then stores the result of add in place of the two parameters
    ISA: A-type
    Example: **add**
    Visualization of the stack



11. **sub** takes the two values stored at the top of the stack, subtracts the values, and then stores the result of sub at the top of the stack
    ISA: A-type
    Example: **sub**
    Visualization of the stack

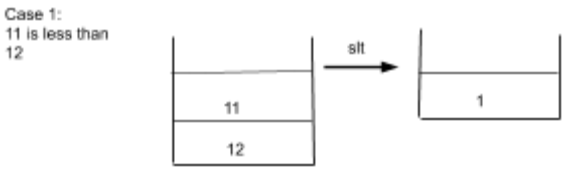


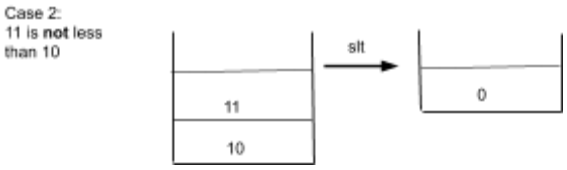
12. **lsl** shifts the value at the top of the stack, logically shifts by the shamt (signed), and replaces the value it operated it on with its result
    ISA: A-type
    Example: **lsl**
    Visualization of the stack



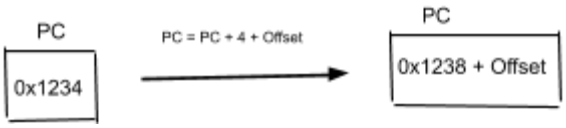
13. **lsr** shifts the value at the top of the stack, logically shifts it by the shamt (signed), and replaces the value it operated on with its result
    ISA: A-type
    Example: **lsr**
    Visualization of the stack

14. **slt** compares the two top-most values of the stack. If the top value of the stack is less than the second value, then return 1. Otherwise, return 0.
    ISA: A-type
    Example: **slt**
    Visualization of the stack

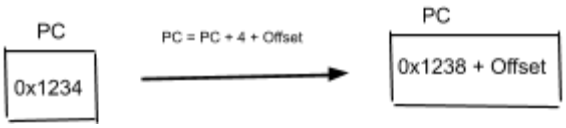




15. beq compares the two top-most values of the stack. If the two values are equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **beq LABEL**
Visualization of Stack



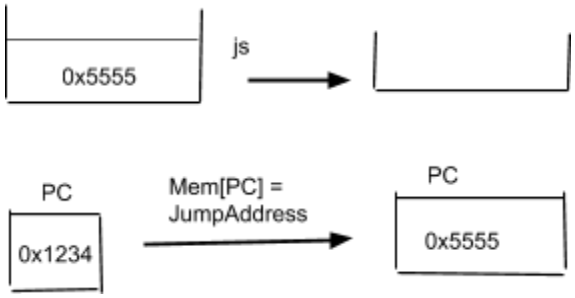
16. bne compares the values compares the top-most values of the stack. If the two values are **not** equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **bne LABEL**
Visualization of stack



17. j causes the memory execution to go to the specified location in memory.

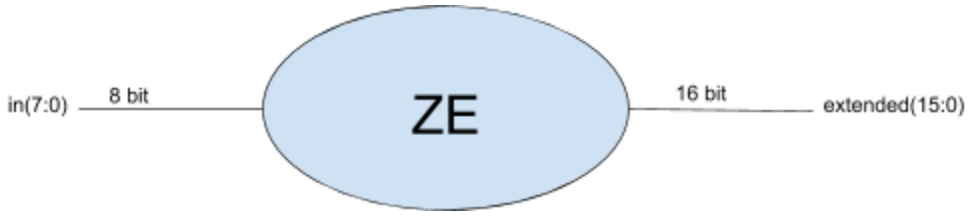
ISA: B-type
Example: **j 0x5555**
Visualization of stack

18. js causes the memory execution to jump to the address that is on the top of the stack (assume that there is an address at the top of the stack)
ISA: B-type
Example: **js**
Visualization of stack



Appendix D: System Components:

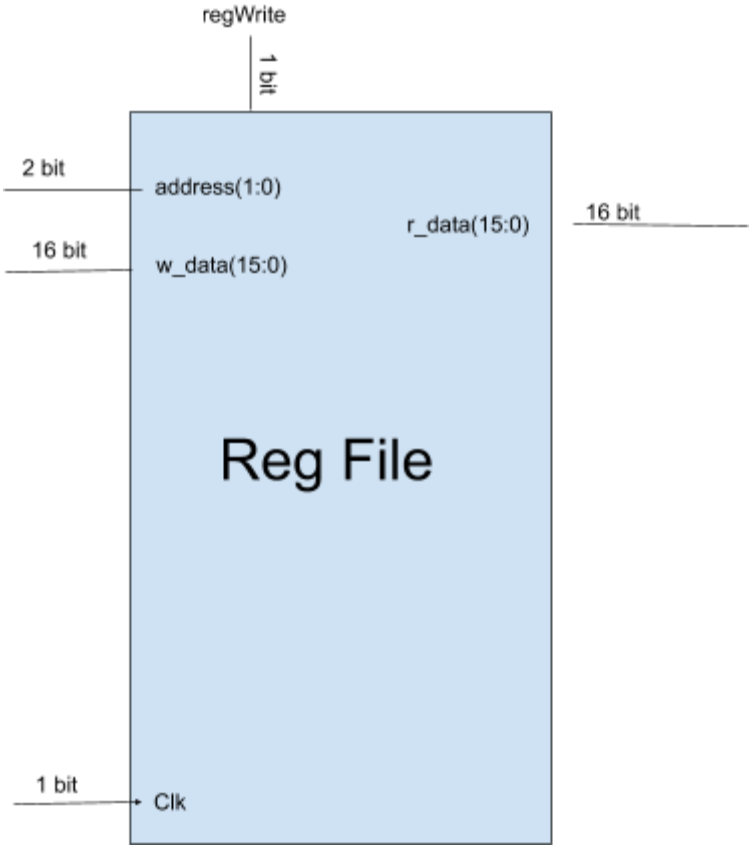
- **Zero Extender:** Zero extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with zeros
    - **Implements RTL Code:** ZE()
    - **Control signal:** None
    - **Input/Output signals:** in(7:0),extended(15:0)
    - **Hardware Implementation:** Assign the eight least significant bits of extended(15:0) to in(7:0). Assign the eight most significant bits to 0.
    - **Unit Testing:** This unit was tested by inputting various in(7:0) and checking that each extended(15:0) matched its input.



- **Sign Extender:** Sign extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with its 8th bit value.
    - **Implements RTL Code:** SE()
    - **Control signal:** None
    - **Input/Output signals:** in(7:0),extended(15:0)
    - **Hardware Implementation:** Assign the eight least significant bits of extended(15:0) to in(7:0). Assign the eight most significant bits to in(7).
    - **Unit Testing:**The test cases are as follows:
        - in(7:0) has a MSB of 0. We check that in(7:0) is equivalent to extended(15:0).
        - in(7:0) has a MSB of 1. We calculate the expected extended(15:0) and compare to the actual result.

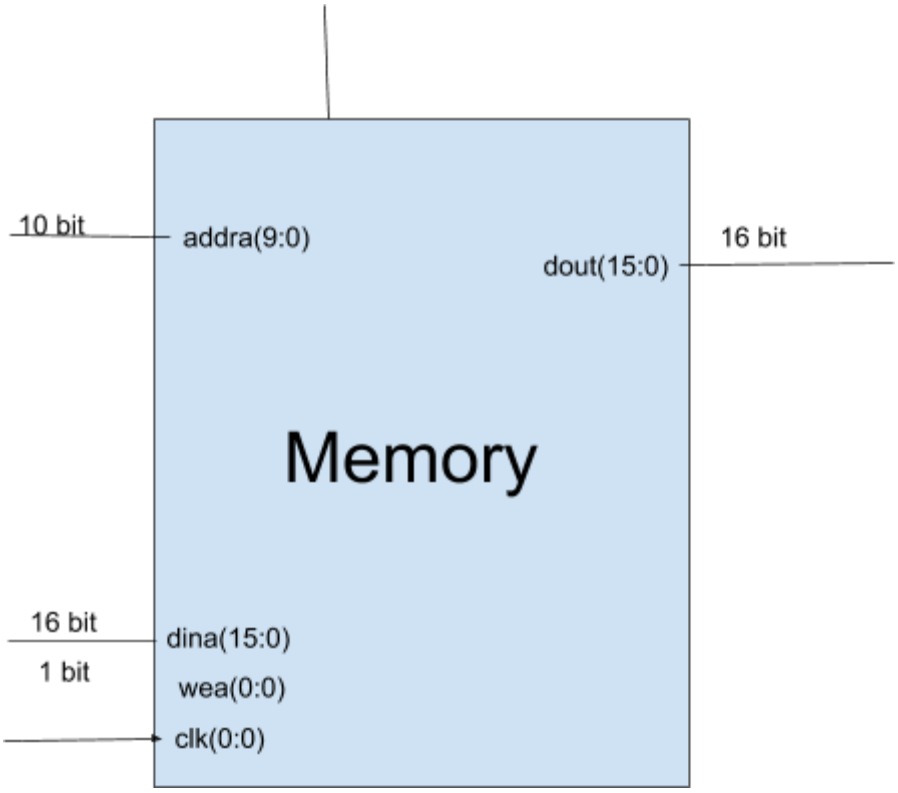
- **Register File (4-bit):** The register file contains 4 registers that refer to commonly used registers such as $V0 (return value), $sp (stack pointer for memory), $gp (global pointer), and $zero (zero register) that allows for reading and writing.
    - **Implements RTL Code:** Reg[]
    - **Control signal:** MemToReg, regWrite

- **Input/Output signals:** clk, address(1:0),w_data(15:0),r_data(15:0)
- **Hardware Implementation:**
- **UnitTesting:** The test cases are as follows:
  - Write to special registers v,sp, and gb and check that r_data matches accordingly
  - Read z register and ensure that value is 0
  - Write to the z register and ensure that it always outputs 0



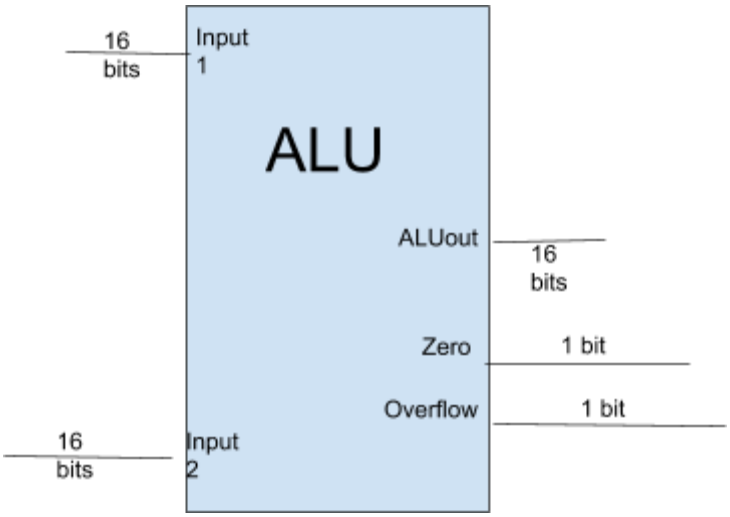
- **Memory:** allows information to be written and read within a range of addresses. See page 6 for how those addresses are allocated.
  - **Implements RTL Code:** Mem[]
  - **Control signal:** MemRead, MemWrite,inst,wea
  - **Input/Output signals:** addra(9:0),dina(15:0),dout(15:0),clk(0:0)
  - **Hardware Implementation:** Use IP(Core Generator and Architecture Wizard) to generate a v6.3 block memory component with a write width of 16, write depth of 1024, and read depth of 1024. The operating mode is write first and is always enabled. A small_memory.coe file is available with arbitrary values stored.
  - **Unit Testing:** The test cases are as follows:
    - Read and checks various values from the memory file that were initialized in the .coe file (Reading without writing)

- Write and checks various values to addresses that were not written to in the .coe file
- Rewrites and checks the value of an address that was already written to in the .coe file



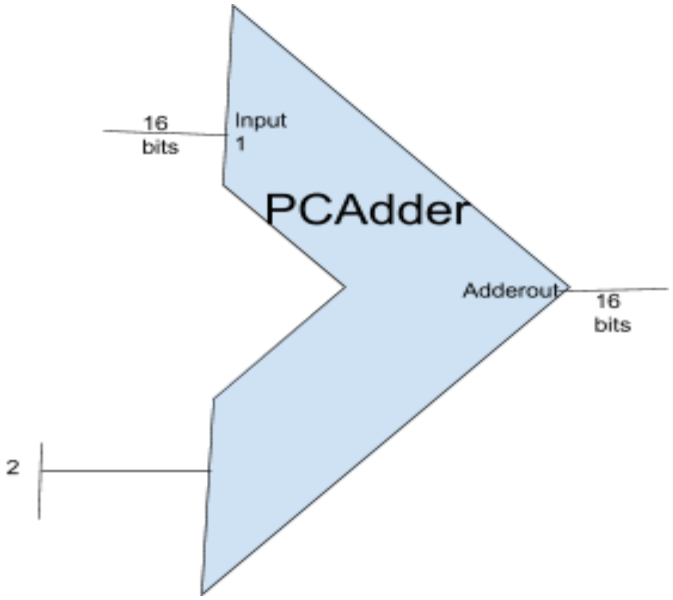
- **Register:** A component that stores the 16-bit value while another value is being processed on datapath
  - **Implements RTL Code:** PC, ALUout, A, B, C, D, E, IR, MDR
  - **Control signal:** PCWrite, PCWriteCond, PCsource, IRWrite
  - **Input/Output signals:** clk, regWrite, w_data(15:0), r_data(15:0)
  - **Hardware Implementation:** If regWrite is enabled, save value of w_data to reg variable. If regWrite is disabled, store value of reg variable to r_data. A register commits these actions on the negative clock edge.
  - **Unit Testing:** The test cases are as follows:
    - Write an initial value to a register and check r_data
    - Rewrite a different value to the register and check r_data to ensure that there was a change
    - Check r_data again to ensure that the register holds data over time
    - Check that a value larger than 16-bits will result in a fail

- **ALU:** The ALU component takes in two 16-bit inputs, and arithmetically computes a single 16-bit result from using functions: “add”, “subtract”, “or”, and “zero detect”.
    - **Implements RTL Code:** +, -, ||, isZero()
    - **Control signal:**
        - ALUsrcA - 2 bits
        - ALUsrcB - 2 bits
        - ALUOp - 2 bits
    - **Input/Output signals:** Input1, Input2, Op/ Zero, Overflow, ALUout
    - **Hardware Implementation:** The ALU has 3 inputs: a, b, and op. The inputs a and b are the values that we would like to operate on while op is the value that determines what operation we shall perform. Once the operation is performed, the value is placed into a 17 bit register. The most significant bit of which is the overflow detector. The remaining bits are the result. The zero detector takes this result and or’s each bit to see if there are any 1’s in the result. If there is, the zero value is 0, and if not it is 1.
    - **Unit Testing:** This unit was tested by selecting two random 16 bit inputs and performing each of the operations that the ALU could perform on these inputs. The output was compared against the expected value. As for the zero and overflow values, they had special inputs to test which would guarantee that they output 1. Subtraction of equal numbers provided the test for the zero output (as well as having b be smaller than a and performing a set less than operation which outputs 0 in that case). Overflow was tested via inputting two 16 bit values which when added would result in a 17 bit output.



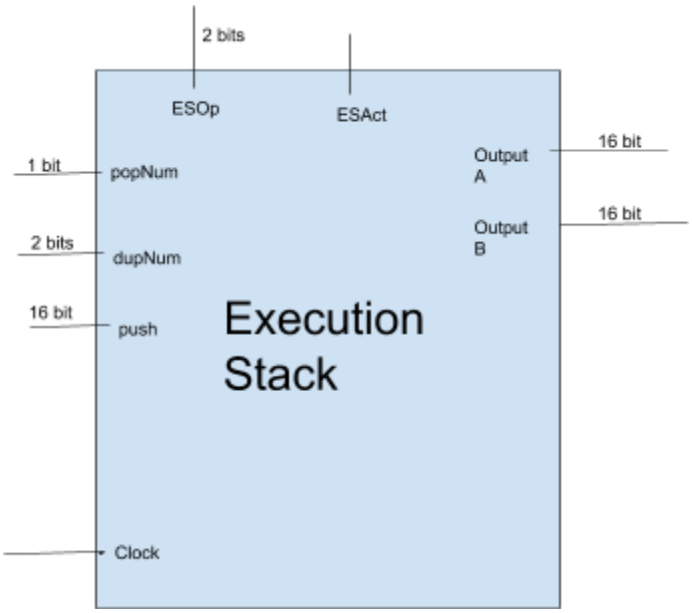
- **PCAdder:** The ALU component takes in two 16-bit inputs (from PC) and adds 2 for proceeding to the next instruction.
    - **Implements RTL Code:** + 2
    - **Control signal:**
        - N/A

- **Input/Output signals:** Input1 / AdderOut
- **Hardware Implementation:** This adder is simply used to add ‘2’ to the PC so that we can increment to the next instruction. This is done by taking in the current PC value and adding 2 to it. The value 2 is hardwired in while PC is the input value. The output is our new PC value.
- **Unit Testing:** This unit was tested by putting in a 16 bit value and adding 2 to it. The output was compared to the expected value.

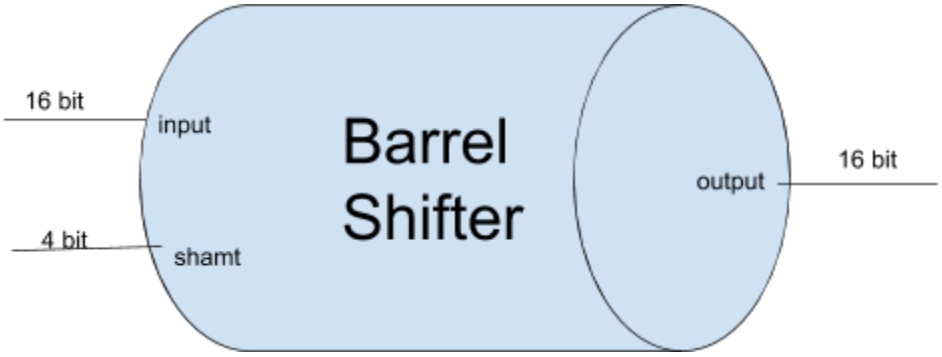


- **Execution Stack** is a component that serves as workspace with temporary data manipulation. It is capable of popping off four 16-bit values at once, duplicating 4 16-bit values, or pushing a single 16-bit value onto the stack.
  - **Implements RTL Code:** Pop, Push, Top, Flip
  - **Control signal:**
    - ESOp - 2 bits, ESAct - 1 bit
  - **Input/Output signals:** push, popNum, dupNum, flip / OutputA, OutputB, OutputC, Output
  - **Hardware Implementation:** The execution stack has the following inputs: 16-bit pushVal, 2-bit popNum and dupNum, 2-bit ESOp and a clock. There are 4 16-bit output registers, and there is an internal 32 register file that can contain 16-bit entries as well as an integer that references the Top of Stack (tos). The pushVal will be placed in reference to the tos, and then the tos will be updated. The popNum will determine how many 16-bit values are placed into the different outputs, and the tos will be updated according to said number. The dupNum will determine how many registers according to the tos will be duplicated, resulting in another update of the tos. There is also a flip function that will reverse the top two registers according to the tos (this does not update the tos). The function that will be executed is determined by the 2-bit ESOp, which is the controller for the execution stack.

- **Unit Testing:** The execution stack was tested through a series of conditions based upon the four operation. The test was based on the OutA value after a popNum=0 condition. First the stack was filled by a series of 33 pushes (This also tests Out of Bounds errors). Then the stack tried to duplicate for each of the four values of dupNum (Note that this was 8 different conditions to include Out of Bounds errors). The stack was then brought down to a total of 4 registers filled where the popNum conditions are tested with their respective Out of Bounds errors. Last, three registers are filled, and the flip operation is tested (The bottom of the stack should not have changed, but the top two should be flipped).



- **Barrel Shifter:** The Barrel Shifter takes in a 16-bit input, shifts the input by a given 4-bit shamt amount, that then produces a 16-bit output
  - **Implements RTL Code:** SL(), SR()
  - **Control signal:**
    - ShiftDirection - 1 bit
  - **Input/Output signals:** input, shamt / output
  - **Hardware Implementation:** The shifter takes in two items, the input value and the shift amount (shamt). The shamt is a signed 4 bit integer and so can be from -8 to 7. The positive values correspond to a left shift as that is considered to occur more often than a right shift. Negative values are a right shift. The output is a 16 bit integer which is the input shifted by the specified amount.
  - **Unit Testing:** This unit was tested by inputting a random 16 bit value and shifting by assorted positive and negative values. The output was checked to see if the 1 bits were in the correct place and that 0's had been shifted in.



- **Control Unit:** Provides the correct control signals to the appropriate components.
    - **Implements RTL Code:** N/A
    - **Control Signal Descriptions:**
        - popAmt
            - provides the correct number of pops to Execution Stack
        - Flip
            - Tells the execution stack whether or not to flip (not part of ESOp?)
        - ESOp
            - Tells the execution stack which operation to perform
        - ESAct
            - Tells the execution when to execute an operation like an enable bit
        - ALUSrcB
            - Specifies whether or not the second input to the ALU is extended or from a register
        - ALUOp
            - Specifies which operation the ALU performs
        - PCSrc
            - Specifies whether the PC is set to the next instruction address or a branched address
        - PushSrc
            - Specifies whether the Execution stack pushes a value from a register, memory, the ALU, the barrel shifter(shifter), or the zero extender
        - ShiftSrc
            - Specifies whether the Shifter shifts an immediate by 2 or by a different amount.
        - RegWrite
            - Tells the register file whether or not to write to one of its registers
        - IorD
            - Determines whether the address of the memory component is from the PC or the ALU
        - wea
            - Tells the memory whether or not to write to the address

- Reset
  - Sets control unit to default values
- PCwrite
  - Enables writing to PC when we DO NOT branch
- BNECond
  - Enables writing to PC when we perform a bne instruction.
- BEQCond
  - Enables writing to PC when we perform a beq instruction.

- **Output signals: all controls above**
- **Input signals: reset, opcode, funct, dampt**
- **Unit Testing:** This unit was tested by manually setting the appropriate opcode, funct, and dampt for all 18 instructions. The appropriate control signal values were then checked for correctness. Reset was tested by setting it to 1 and seeing if changing the other inputs produced a different result from expected.