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 found on page 3. The semantics of how the stack works with these operations can be found in the *Instruction Syntax and Semantics* section on page 5. 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.

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 realprime code to reflect these changes

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	10 bits	2 bit
OPCODE	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.

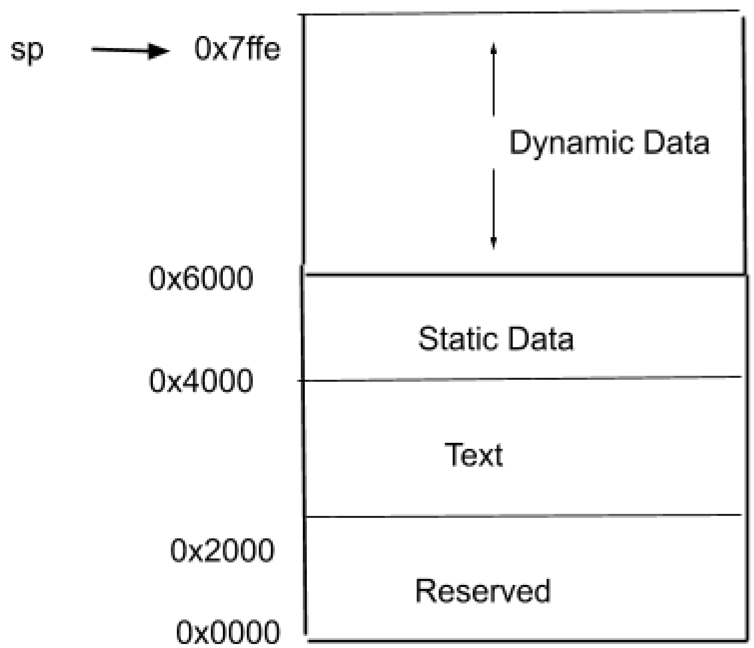
Local Variable Convention:

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:

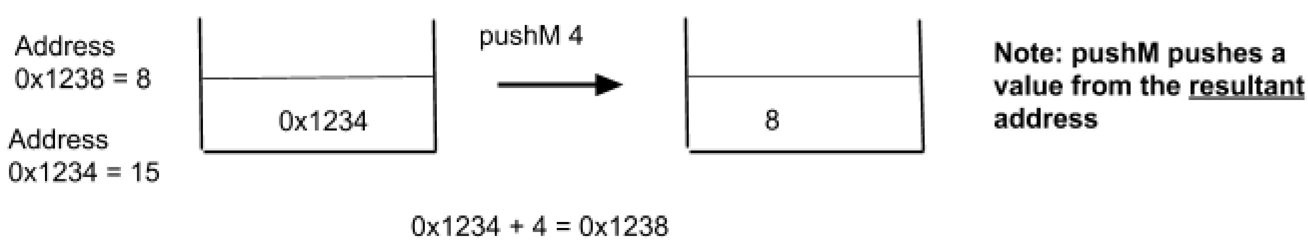
We have chosen the size of our register stack to be constructed out of 64 16-bit available registers. This decision was made because this is the maximum size that the FPGA board allows.

Memory Allocation:

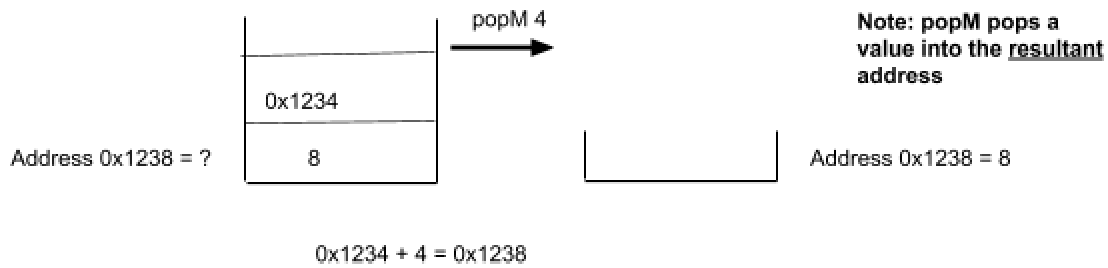


Instruction syntax and semantics:

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



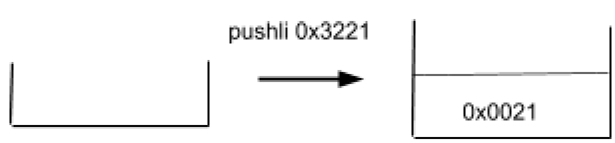
2. **popM** takes the 16-bit address that is on the top of the stack, adds an immediate to the address, and stores the value below that into the resultant address
   ISA: C-type
   Example: **popM 4**
   Visualization of the stack



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

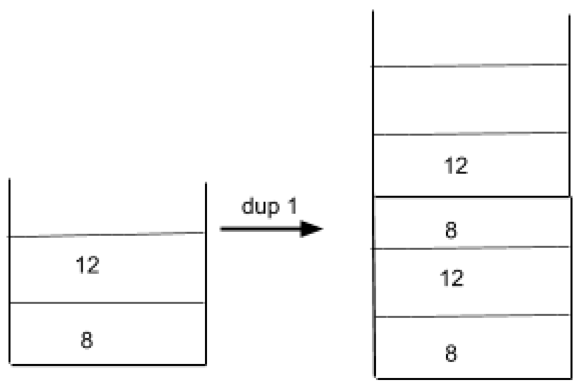
4. **popR** stores a 16 bit value from the top of the stack into the specified register. The value is then popped off the stack.
   ISA: C-type
   Example: **popR $sp**
   Visualization of the stack

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

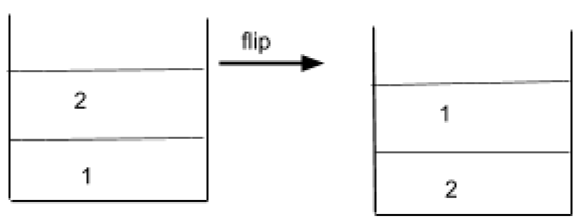


6. **pushui** takes the upper 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack.
   ISA: C-type
   Example: **pushui 0x32**
   Visualization of the stack

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**

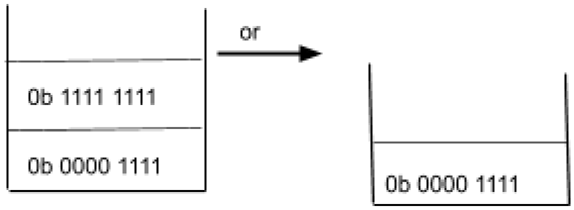


8. **flip** takes the two topmost values in the stacks and reverses their order on the stack.
   ISA: A-type
   Example: **flip**
   Visualization of stack

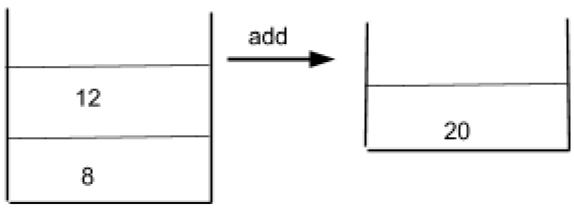


9. **or** looks at the two topmost values of the stack and performs the bitwise ‘or’ operation. The result is stored at the top of the stack.
   ISA: A-type

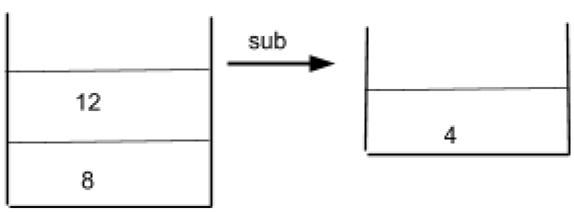
Example: **or**
Visualization of stack



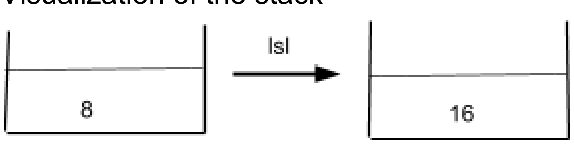
10. **add** takes the two values stored at the top of the stack, adds the values, and then stores the result of add in place of the two parameters
ISA: A-type
Example: **add**
Visualization of the stack



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

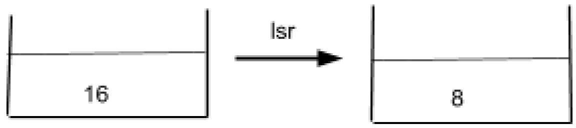


12. **lsl** shifts the value at the top of the stack, logically shifts it to the left once, and replaces the value it operated it on with its result
ISA: A-type
Example: **lsl**
Visualization of the stack

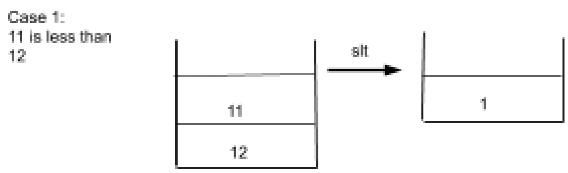


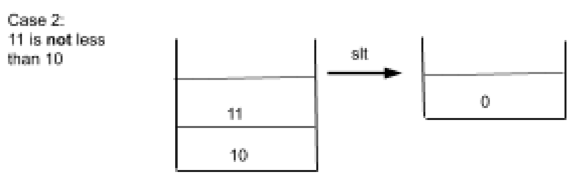
13. **lsr** shifts the value at the top of the stack, logically shifts it to the right once, and replaces the value it operated on with its result

ISA: A-type
Example: **lsr**
Visualization of the stack

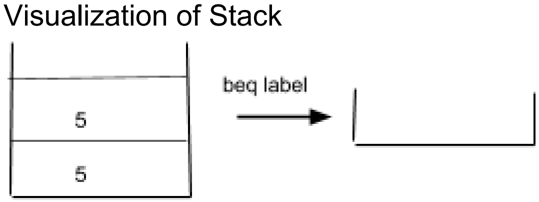


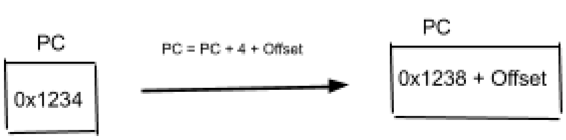
14. slt compares the two top-most values of the stack. If the top value of the stack is less than the second value, then return 1. Otherwise, return 0.
ISA: A-type
Example: **slt**
Visualization of the stack



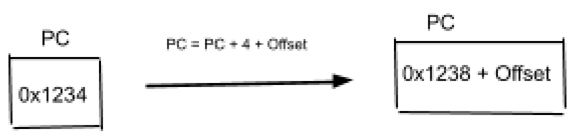


15. beq compares the two top-most values of the stack. If the two values are equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **beq LABEL**
Visualization of Stack

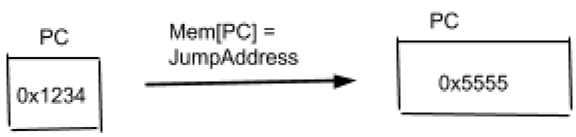




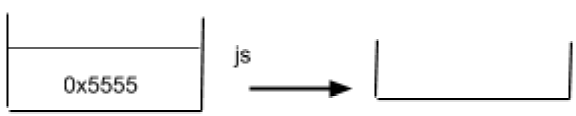
16. **bne** compares the values compares the top-most values of the stack. If the two values are **not** equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **bne LABEL**
Visualization of stack

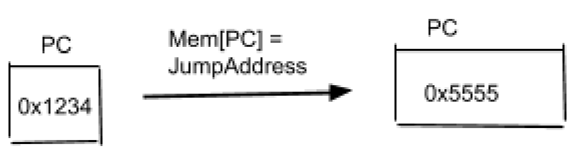


17. **j** causes the memory execution to go to the specified location in memory.
ISA: B-type
Example: **j 0x5555**
Visualization of stack



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





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 0x26	0b 0010 0010 0110 0000	Push RETURN
0x200E	pushUI 0x20	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

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

0x205E	js	0b 1111 0000 0000 0000	Off, jumps to RA
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	

Common Operations:

1. **Conditional Statements:**
    a. *Less than*

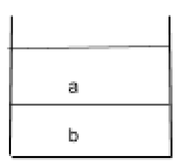
C code:
```
if(a < b){
    a = a + b
}
else{
    b = a + b
}
```

Assembly code:
```
    push $sp
    pushM 0
    push $sp
    pushM -2
    add

    push $sp
    pushM
    push $sp
    pushM -2
    slt
    pushR $zero
    bne DOIT
    push $sp
    popM -2

DOIT:
    push $sp
    popM 0
```

ASSUMPTION;
Initial stack before if statement:



b. Greater than or equal

C code:

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

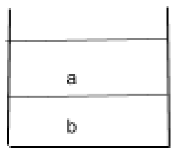
Assembly code:

(*same as less than*)

slt
pushR $zero
beq DOIT

(*same as less than*)

ASSUMPTION;
Initial stack before if statement:



c. Less than or equal

C code:

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

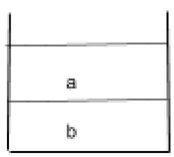
Assembly code:

(*same as less than*)

flip
slt
pushR $zero
beq DOIT

(*same as less than*)

ASSUMPTION;
Initial stack before if statement:



d. Greater than

C code:

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

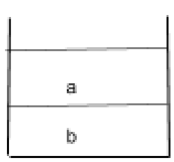
Assembly code:

(*same as less than*)

flip
slt
pushR $zero
bne DOIT

(*same as less than*)

ASSUMPTION;
Initial stack before if statement:

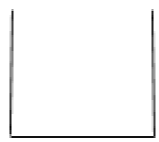


2. Looping

a. While loop

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

ASSUMPTION;
Initial stack looks like:



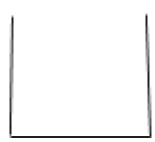
Assembly code:
```
	push $zero
	push $sp
	dup 1
	popM 0
	popM -2

LOOP:
	push $zero
	push $sp
	pushM 0
	pushLi 10
	slt
	beq END:
	push $sp
	pushM 0
	dup 0
	push $sp
	pushM -2
	add
	push $sp
	popM -2
	pushLi 1
	add
	push $sp
	popM 0
	j LOOP
```

b. For loop

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

ASSUMPTION;
Initial stack looks like:



Assembly code:
```
	push $zero
	push $sp
	dup 1
	popM 0
	popM -2

LOOP:
	push $zero
	push $sp
	pushM 0
	pushLi 10
	slt
	beq END:
	push $sp
	pushM 0
	dup 0
	push $sp
	pushM -2
	add
	push $sp
	popM -2
	pushLi 1
	add
	push $sp
	popM 0
	j LOOP
```

3. Procedure Calling

a. Caller to Callee

C code:

```
(*some code*)
a = getGCD(a,b);
```

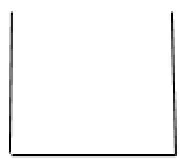
Assembly code:

```
        push $sp
        dup 0
        dup 0
        pushLi TEMP
        pushUi TEMP
        or
        pushM 0
        pushM -2

        push $sp
        pushLi 4
        add
        pop $sp
        j GCD
TEMP:
        push $sp
        pushLi -4
        add
        pop $sp

        push $V0
        pushM 0
        popM 0
```

ASSUMPTION;
Initial stack looks like:



b. Callee to Caller

C code:

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

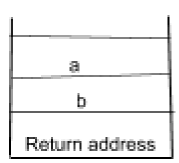
Assembly code:

```
        push $sp
        popM 0
        push $sp
        popM -2

        (*DO SOME CODE*)

        push $sp
        pushM 0
        push $V0
        popM 0
        js
```

ASSUMPTION;
Initial stack looks like:



RTL:

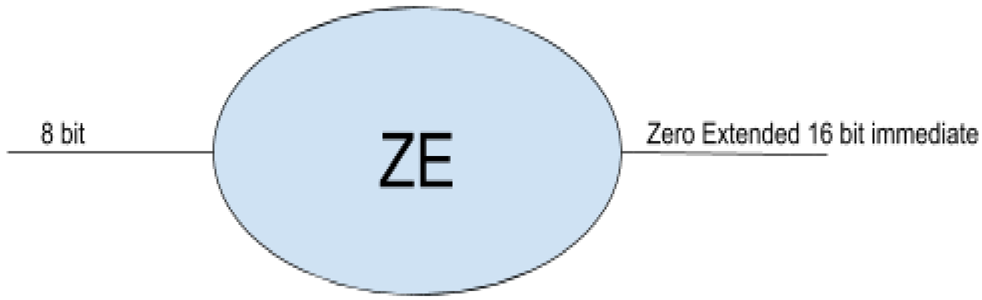
Step	Push/Pop M	Push/Pop R	Arithmetic/ Logic	beq/bne	j	js	slt	lsr/ lsl	flip	dup	pushUI/LI
Inst Fetch	newPC = PC + 2 PC = newPC Inst = Mem[PC]										
Inst Decode Pop from stack pushLI/ UI done dupAMT fetched	A = pop B = pop									dupAMT = inst[1-0]	UI: push inst[11-4] << 2 LI: push ZE(inst[11-4])

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:
	Memout = Mem[ALUout]
	
	popM:
	Mem[ALUout] = B
		popR:
		Reg[inst[1-0]] = A
		
		pushR: Push reg[inst[1-0]]
			ALUout = A op B	If (A==B) then
				
				PC =PC[15-11] || inst[11-0]
					PC = PC[15-11] || inst[11-0]	Address = pop
						PC = address
							If (A < B)
							Push 1
							
							Else
							Push 0
								A = A shifted by B in the appropriate direction	Push A
									Push B
										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	

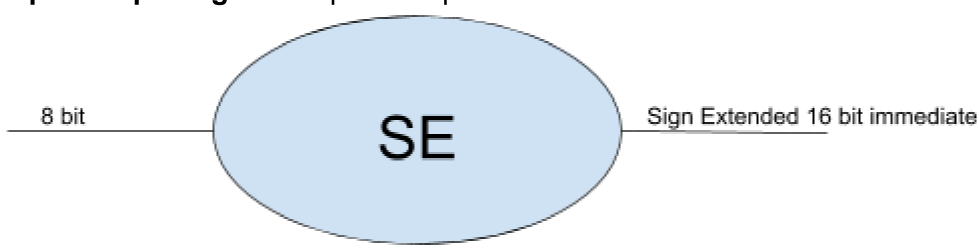
Push to stack Arithmetic / Logic Done pushM done	pushM: push Memout		Push ALUout								

Required Hardware:

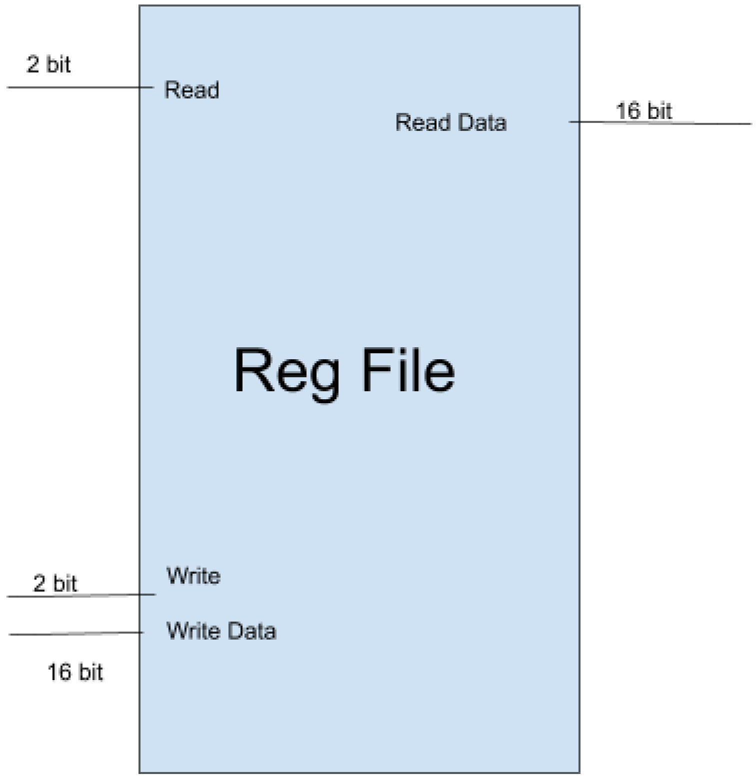
- **Zero Extender:** Zero extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with zeros
    - **Implements RTL Code:** ZE()
    - **Control signal:** None
    - **Input/Output signals:** Input / Output



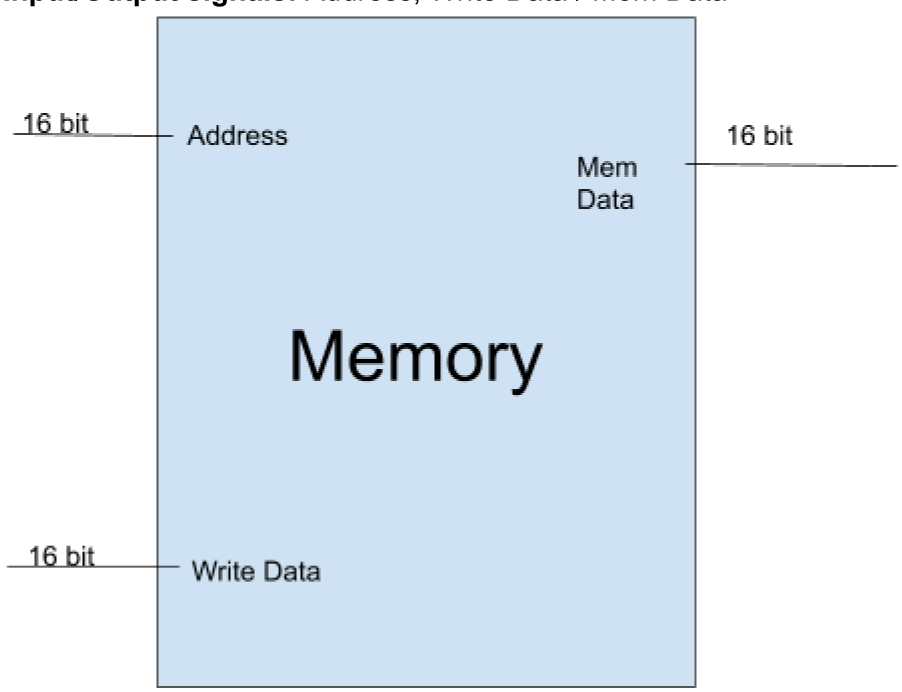
- **Sign Extender:** Sign extender takes an 8 bit immediate and extends it to a 16 bit immediate by filling in its 8 most significant bits with its 8th bit value.
    - **Implements RTL Code:** SE()
    - **Control signal:** None
    - **Input/Output signals:** Input / Output



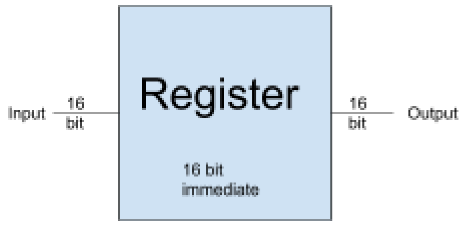
- **Register File (4-bit):** The register file contains 4 registers that refer to commonly used registers such as $V0 (return value), $sp (stack pointer for memory), $gp (global pointer), and $zero (zero register) that allows for reading and writing.
    - **Implements RTL Code:** Reg[]
    - **Control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



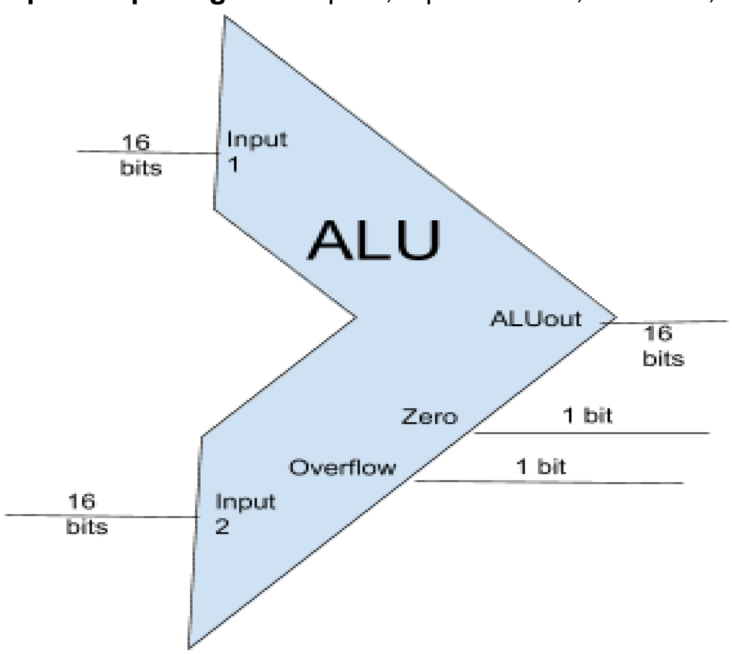
- **Memory:** allows information to be written and read within a range of addresses. See page 6 for how those addresses are allocated.
    - **Implements RTL Code:** Mem[]
    - **Control signal:** MemRead, MemWrite,inst
    - **Input/Output signals:** Address, Write Data / Mem Data



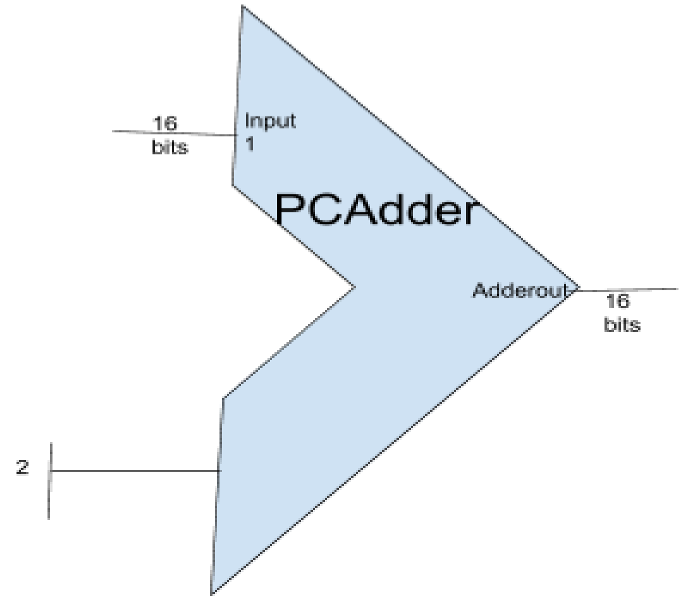
- **Register:** A component that stores the 16-bit value while another value is being processed on datapath
  - **Implements RTL Code:** PC, ALUout, A, B, C, D, E, IR, MDR
  - **Control signal:** PCWrite, PCWriteCond, PCsource, IRWrite
  - **Input/Output signals:** input / output



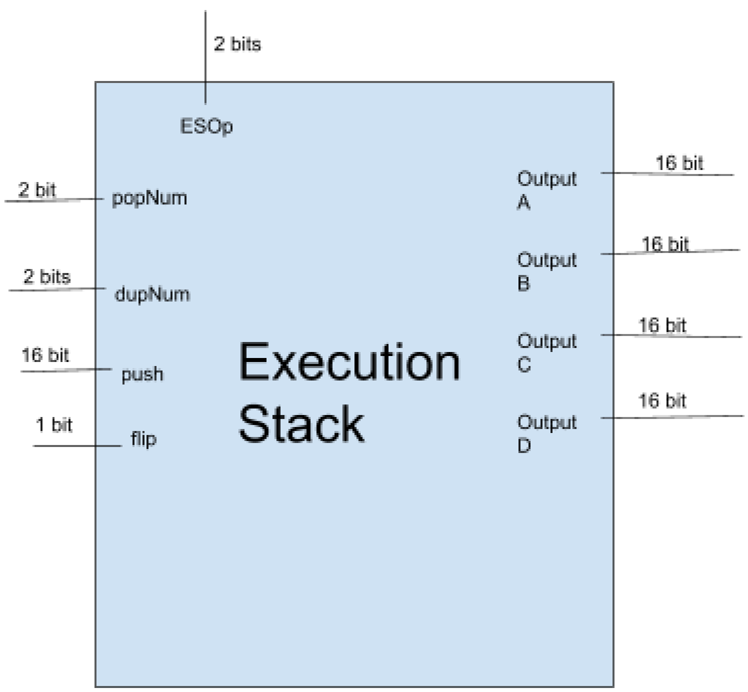
- **ALU:** The ALU component takes in two 16-bit inputs, and arithmetically computes a single 16-bit result from using functions: “add”, “subtract”, “or”, and “zero detect”.
  - **Implements RTL Code:** +, -, ||, isZero()
  - **Control signal:**
    - ALUsrcA - 2 bits
    - ALUsrcB - 2 bits
    - ALUOp - 2 bits
  - **Input/Output signals:** Input1, Input2 / Zero, Overflow, ALUout



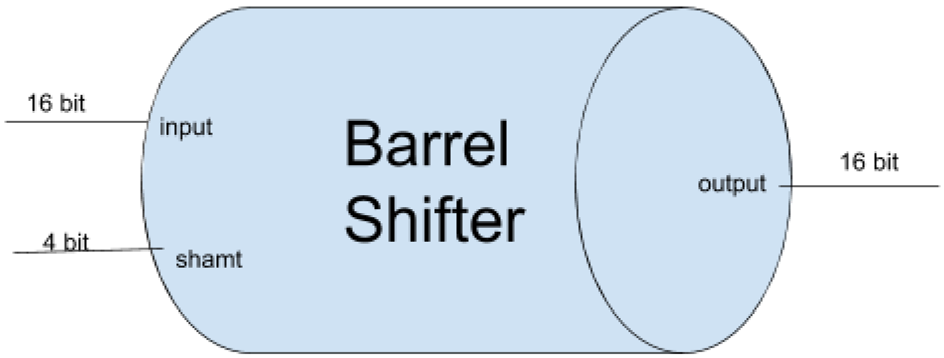
- **PCAdder:** The ALU component takes in two 16-bit inputs (from PC) and adds 2 for proceeding to the next instruction.
  - **Implements RTL Code:** + 2
  - **Control signal:**
    - N/A
  - **Input/Output signals:** Input1 / AdderOut



- **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 push a single 16-bit value onto the stack.
    - **Implements RTL Code:** Pop, Push, Top, Flip
    - **Control signal:**
        - ESOp - 2 bits
    - **Input/Output signals:** push, popNum, dupNum, flip / OutputA, OutputB, OutputC, Output



- **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

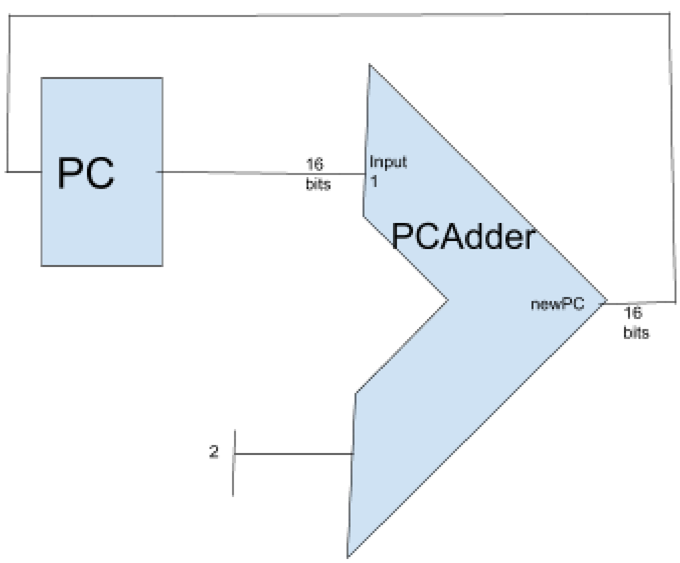


RTL Verification:

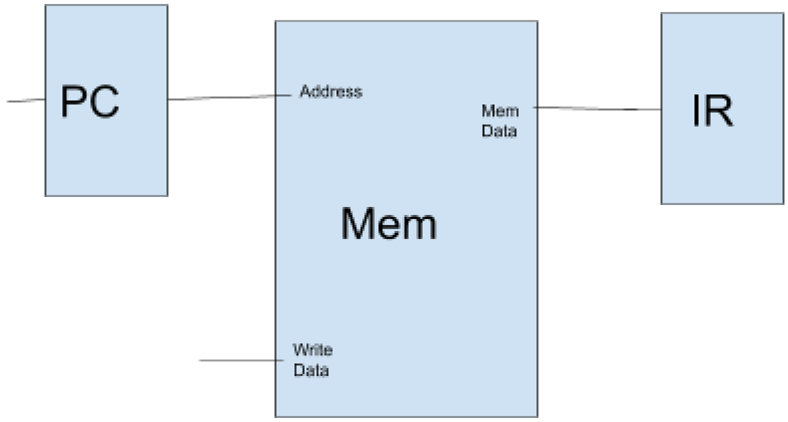
We verified our register transfer language by simulating the implementation of each instruction through the described above components. Below, we will show how we simulated add:

__Add__

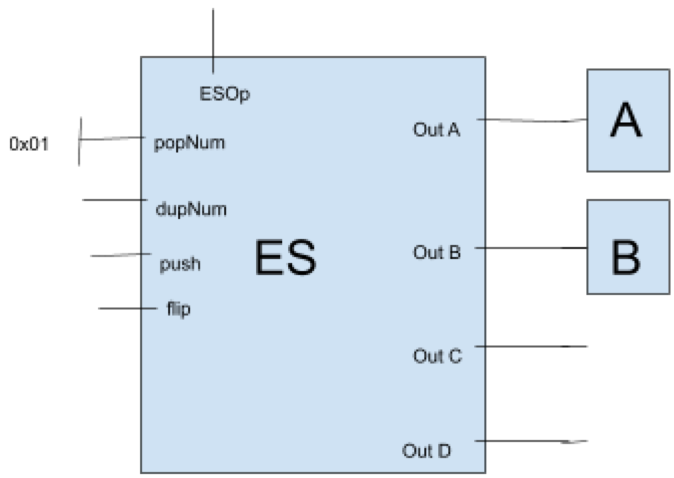
1. ***newPC = PC + 2***
2. ***PC = newPC***



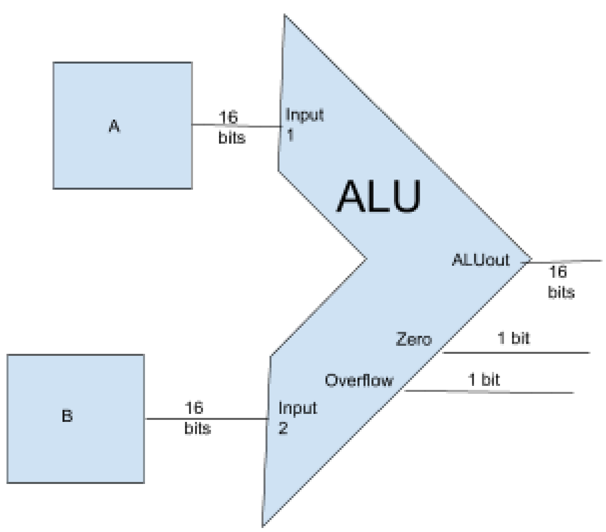
3. ***Inst = Mem[PC]***



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



6. ***ALUout = A+B***



7. Push ALUout