Documentation - Team SV - Toaster Troop

- Title page
- T.O.C.
- executive summary

List of Commands:

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

Reserved Memory Addresses:

1. Stack Pointer - SP = 0b0001 = 0x0001
2. Return Value - V0 = 0b0010 = 0x0002
3. Global Pointer - GP = 0b0011 = 0x0003
4. Zero Register - zero = 0b0000 = 0x0000

just one byte each?

Pick a design, stick with it!

Possible ISA designs:

Items involving immediates: C-type

4 bits	8 bits	4 bits
OPCODE	IMMEDIATE	TBD

Arithmetic Items: A-Type

4 bits	12 bits
OPCODE	OTHER ?? what is it?

Jumping / Branching: B-Type

4 bits	12 bits
OPCODE	ADDRESS/IMMEDIATE

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

more details, please. provide example

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.

why not on stack?

which order? which first?

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.

:) you can change it later

RelPrime Code:

Address/Label	**Assembly**	**Machine**
0x4000	pushLI 2	0b 0010 0000 0010 0000
0x4002	pushLI addr(m)	0b 0010 dddd dddd[1] 0000
0x4004	pushUI addr(m)	0b 0011 dddd dddd 0000
0x4006	or	0b 0110 0000 0000 0000
0x4008	pop	0b 0001 0000 0000 0000
0x400A	pushLI addr(n)	0b 0010 dddd dddd 0000
0x400C	pushUI addr(n)	0b 0010 dddd dddd 0000
0x400E	or	0b 0110 0000 0000 0000
0x4010	pop	0b 0001 0000 0000 0000
0x4012	pushLI 1	0b 0010 0000 0001 0000
0x4014 / LOOP	pushLI 0x28	0b 0010 0010 1000 0000
0x4016	pushUI 0x40	0b 0011 0100 0000 0000
0x4018	pushLI addr(m)	0b 0010 dddd dddd 0000
0x401A	pushUI addr(m)	0b 0011 dddd dddd 0000
0x401C	or	0b 0110 0000 0000 0000
0x401E	push	0b 0000 0000 0000 0000
0x4020	pushLI addr(n)	0b 0010 dddd dddd 0000
0x4022	pushUI addr(n)	0b 0011 dddd dddd 0000
0x4024	or	0b 0110 0000 0000 0000
0x4026	push	0b 0000 0000 0000 0000
0x4028	j GCD	0b 1110 0000 0101 1011
0x402A	pushLI addr (V0)	0b 0010 0000 0010 0000
0x402C	pushUI addr(V0)	0b 0011 0000 0010 0000

[1] d denotes that the bit values are determined by an unknown address value

MAKE IT KNOWN!

where is input?

0x402E	or	0b 0110 0000 0000 0000
0x4030	push	0b 0000 0000 0000 0000
0x4032	bne DONE1	0b 1101 0000 0000 1011
0x4034	pushLI addr(m)	0b 0010 dddd[2] dddd 0000
0x4036	pushUI addr(m)	0b 0011 dddd dddd 0000
0x4038	or	0b 0110 0000 0000 0000
0x403A	push	0b 0000 0000 0000 0000
0x403C	pushLI 1	0b 0010 0000 0001 0000
0x403E	add	0b 0111 0000 0000 0000
0x4040	pushLI addr(m)	0b 0010 dddd dddd 0000
0x4042	pushUI addr(m)	0b 0011 dddd dddd 0000
0x4044	or	0b 0110 0000 0000 0000
0x4046	pop	0b 0001 0000 0000 0000
0x4048	j LOOP	0b 1110 0000 0001 0100
0x404A / DONE1	pushLI addr(m)	0b 0010 dddd dddd 0000
0x404C	pushUI addr(m)	0b 0011 dddd dddd 0000
0x404E	or	0b 0110 0000 0000 0000
0x4050	push	0b 0000 0000 0000 0000
0x4052	pushLI addr(v0)	0b 0010 0000 0010 0000
0x4054	pushUI addr(v0)	0b 0011 0000 0010 0000
0x4056	or	0b 0110 0000 0000 0000
0x4058	pop	0b 0001 0000 0000 0000
0x405A[3]	js	0b 1111 0000 0000 0000

[2] d denotes that the bit values are determined by an unknown address value

[3] This is the last instruction in the relPrime program

0x405C / GCD[4]	pushLI addr(b)	0b 0010 dddd dddd[5] 0000
0x405E	pushUI addr(b)	0b 0011 dddd dddd 0000
0x4060	or	0b 0110 0000 0000 0000
0x4062	pop	0b 0001 0000 0000 0000
0x4064	pushLI addr(a)	0b 0010 dddd dddd 0000
0x4066	pushUI addr(b)	0b 0011 dddd dddd 0000
0x4068	or	0b 0110 0000 0000 0000
0x406A	pop	0b 0001 0000 0000 0000
0x406C	pushLI addr(b)	0b 0010 dddd dddd 0000
0x406E	pushUI addr(b)	0b 0011 dddd dddd 0000
0x4070	or	0b 0110 0000 0000 0000
0x4072	push	0b 0000 0000 0000 0000
0x4074	pushLI addr(a)	0b 0010 dddd dddd 0000
0x4076	pushUI addr(a)	0b 0011 dddd dddd 0000
0x4078	or	0b 0110 0000 0000 0000
0x407A	push	0b 0000 0000 0000 0000
0x407C	bne LOOP2	0b 1101 0000 0000 0001
0x407E	js	0b 1111 0000 0000 0000
0x4080 / LOOP 2	pushLI addr(b)	0b 0010 dddd dddd 0000
0x4082	pushUI addr(b)	0b 0011 dddd dddd 0000
0x4084	or	0b 0110 0000 0000 0000
0x4086	push	0b 0000 0000 0000 0000
0x4088	pushLI 0	0b 0010 0000 0000 0000
0x408A	bne DONE2	0b 1101 0000 0010 1011

[4] GCD denotes the beginning of the GCD program
[5] d denotes that the bit values are determined by an unknown address value

0x408C	pushLI addr(b)	0b 0010 dddd dddd[6] 0000
0x408E	pushUI addr(b)	0b 0011 dddd dddd 0000
0x4090	or	0b 0110 0000 0000 0000
0x4092	push	0b 0000 0000 0000 0000
0x4094	pushLI addr(a)	0b 0010 dddd dddd 0000
0x4096	pushUI addr(a)	0b 0011 dddd dddd 0000
0x4098	or	0b 0110 0000 0000 0000
0x409A	push	0b 0000 0000 0000 0000
0x409C	slt	0b 1011 0000 0000 0000
0x409E	pushLI 0	0b 0010 0000 0000 0000
0x40A0	bne CON1	0b 1101 0000 0001 0010
0x40A2	pushLI addr(a)	0b 0010 dddd dddd 0000
0x40A4	pushUI addr(a)	0b 0011 dddd dddd 0000
0x40A6	or	0b 0110 0000 0000 0000
0x40A8	push	0b 0000 0000 0000 0000
0x40AA	pushLI addr(b)	0b 0010 dddd dddd 0000
0x40AC	pushUI addr(b)	0b 0011 dddd dddd 0000
0x40AE	or	0b 0110 0000 0000 0000
0x40B0	push	0b 0000 0000 0000 0000
0x40B2	sub	0b 1000 0000 0000 0000
0x40B4	pushLI addr(b)	0b 0010 dddd dddd 0000
0x40B6	pushUI addr(b)	0b 0011 dddd dddd 0000
0x40B8	or	0b 0110 0000 0000 0000
0x40BA	push	0b 0000 0000 0000 0000
0x40BC	pushLI addr(b)	0b 0010 dddd dddd 0000

[6] d denotes that the bit values are determined by an unknown address value

0x40BE	pushUI addr(b)	0b 0011 dddd dddd[7] 0000
0x40C0	or	0b 0110 0000 0000 0000
0x40C2	pop	0b 0001 0000 0000 0000
0x40C4	J LOOP2	0b 1110 0000 1000 0000
0x40C6 / CON1	pushLI addr(b)	0b 0010 dddd dddd 0000
0x40C8	pushUI addr(b)	0b 0011 dddd dddd 0000
0x40CA	or	0b 0110 0000 0000 0000
0x40CC	push	0b 0000 0000 0000 0000
0x40CE	pushLI addr(a)	0b 0010 dddd dddd 0000
0x40D0	pushUI addr(a)	0b 0011 dddd dddd 0000
0x40D2	or	0b 0110 0000 0000 0000
0x40D4	push	0b 0000 0000 0000 0000
0x40D6	sub	0b 1000 0000 0000 0000
0x40D8	pushLI addr(a)	0b 0010 dddd dddd 0000
0x40DA	pushUI addr(a)	0b 0010 dddd dddd 0000
0x40DC	or	0b 0110 0000 0000 0000
0x40DE	pop	0b 0001 0000 0000 0000
0x40E0	J LOOP2	0b 1110 0000 1000 0000
0x40E2 / DONE2	pushLI addr(a)	0b 0010 dddd dddd 0000
0x40E4	pushUI addr(a)	0b 0011 dddd dddd 0000
0x40E6	or	0b 0110 0000 0000 0000
0x40E8	push	0b 0000 0000 0000 0000
0x40EA	PushLI addr(V0)	0b 0010 0000 0010 0000
0x40EC	pushUI addr(V0)	0b 0011 0000 0000 0000
0x40EE	or	0b 0110 0000 0000 0000

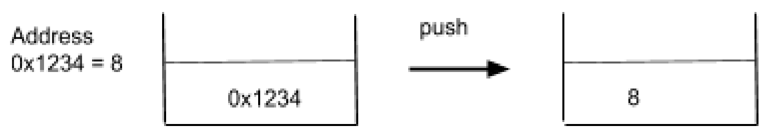
[7] d denotes that the bit values are determined by an unknown address value

0x40F0	push	0b 0000 0000 0000 0000
0x40F2	js	0b 1111 0000 0000 0000

Put before big code

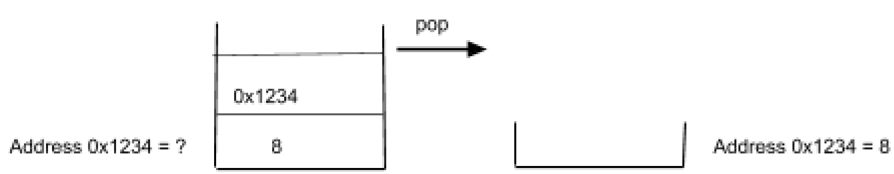
Instruction syntax and semantics:

1. **push** takes the 16-bit address from the top of the stack and returns the value that is stored in that address
   ISA: C-type
   Example: **push**
   Visualization of the stack



nice

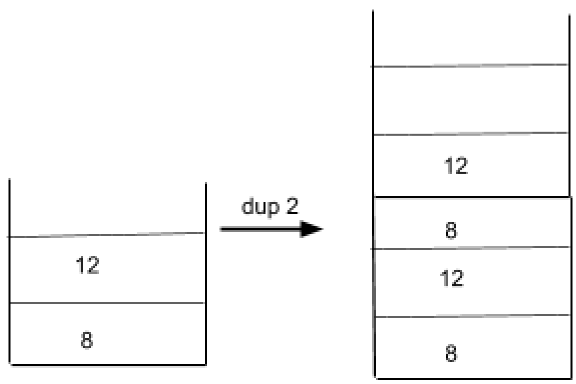
2. **pop** takes the 16-bit address that is on the top of the stack and stores the value below that into that address
   ISA: C-type
   Example: **pop**
   Visualization of the stack



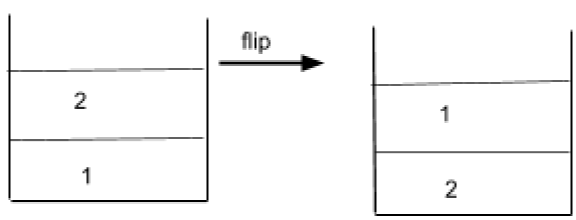
3. **pushli** takes the lower 8 bits of a 16 bit immediate, zero extends it, and stores it on the top of the stack
   ISA: C-type
   Example: **pushli 0x3221**
   Visualization of the stack

4. **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**
   Visualization of the stack

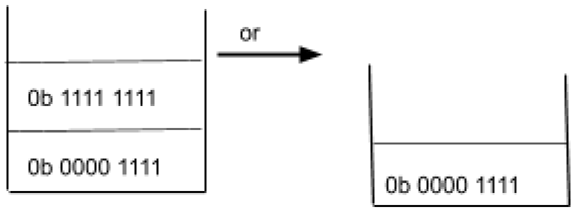
5. **dup** looks at the specified amount of data from the top of the stack, copies the data, and pushes it on to the top of the stack
   ISA: A-type
   Example: **dup 2**



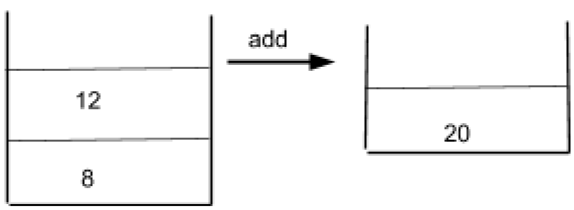
6. **flip** takes the two topmost values in the stacks and reverses their order on the stack.
   ISA: A-type
   Example: **flip**
   Visualization of stack



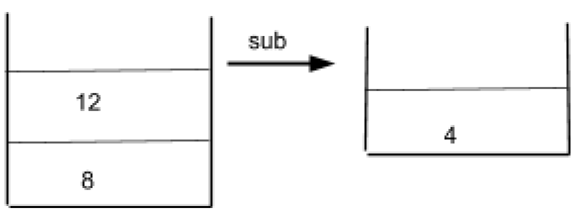
7. **or** looks at the two topmost values of the stack and performs the bitwise ‘or’ operation. The result is stored at the top of the stack.
ISA: A-type
Example: **or**
Visualization of stack



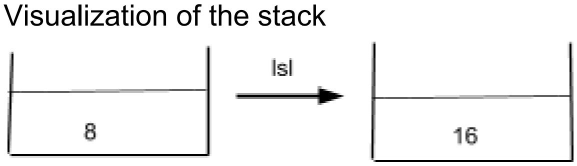
8. **add** takes the two values stored at the top of the stack, adds the values, and then stores the result of add in place of the two parameters
ISA: A-type
Example: **add**
Visualization of the stack



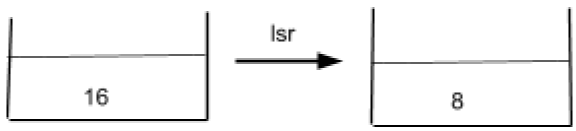
9. **sub** takes the two values stored at the top of the stack, subtracts the values, and then stores the result of sub at the top of the stack
ISA: A-type
Example: **sub**
Visualization of the stack



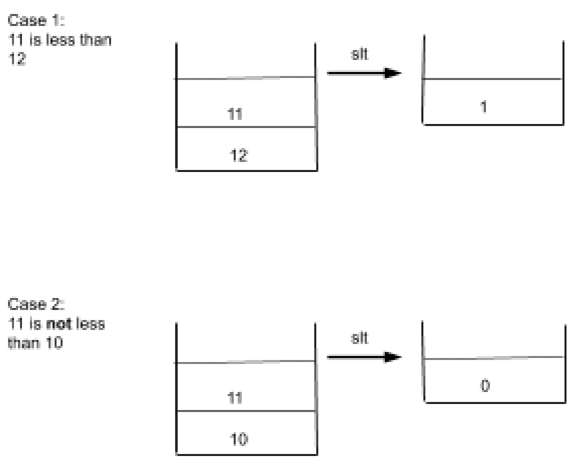
10. **lsl** shifts the value at the top of the stack, logically shifts it to the left once, and replaces the value it operated it on with its result
ISA: A-type
Example: **lsl**
Visualization of the stack



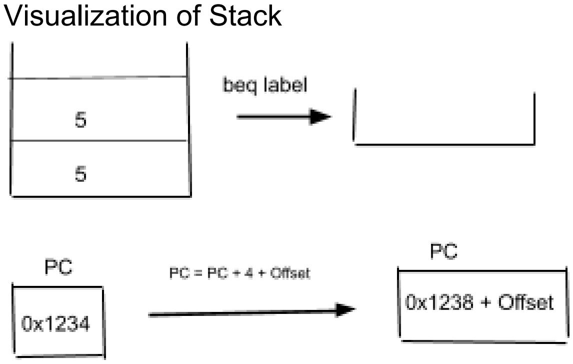
11. lsr shifts the value at the top of the stack, logically shifts it to the right once, and replaces the value it operated on with its result
ISA: A-type
Example: **lsr**
Visualization of the stack



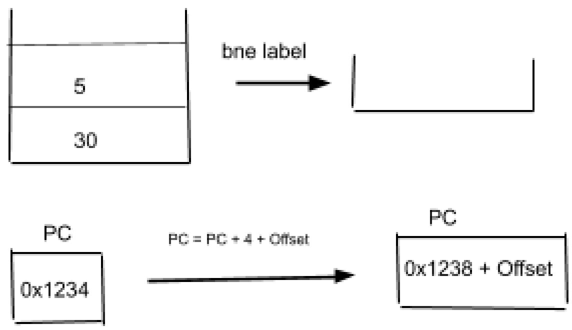
12. slt compares the two top-most values of the stack. If the top value of the stack is less than the second value, then return 1. Otherwise, return 0.
ISA: A-type
Example: **slt**
Visualization of the stack



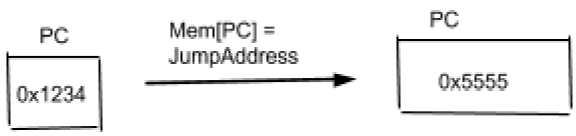
13. beq compares the two top-most values of the stack. If the two values are equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **beq LABEL**
Visualization of Stack



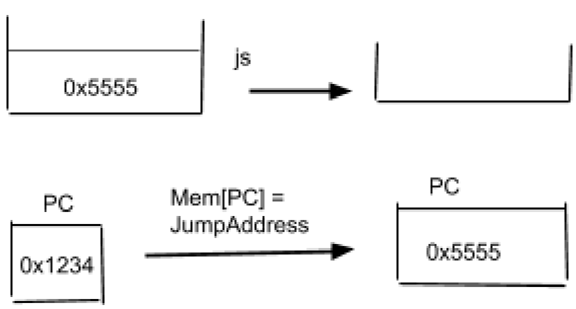
14. bne compares the values compares the top-most values of the stack. If the two values are **not** equal, then the program execution goes to the address referenced by its label.
ISA: B-type
Example: **bne LABEL**
Visualization of stack



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



16. 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



Common Operations:

1. Loading a 16-bit value onto the stack from a memory address
    a. pushLI lower(addr)
    b. pushUI upper(addr)
    c. or
    d. push
2. Popping a 16-bit value on the stack into a memory address
    a. pushLI lower(value)
    b. pushUI upper(value)
    c. or
    d. pushLI lower(addr)
    e. pushUI upper(addr)
    f. or
    g. Pop

Unfinished?
If statement?
Proc call?