Lebron Mk. 2.01

Team: V-2223c-02

Aidan Kirk, Dalton Bumb, Avichal Jadeja, Vishrut Patwari

Link to Google Document
https://docs.google.com/document/d/16eSBUKxdq_vlAOVXJIrhp27hVrMAWL0CEv2cxJjvVYw/edit?usp=sharing

Design

Our design will use a register-stack architecture. There will be three register stacks. A large data stack and two smaller stacks for procedure arguments and return addresses. The data stack is for operations inside of procedures. Our motivation for using the register-stack architecture is to create a stack that strikes an even balance between efficient subroutine calls and local variable storage. The overall goal is to reduce the need for the programmer to access memory. This architecture will enable us to retain simplicity while not compromising on overall efficiency.

Measure Of Performance

- Execution time. The implementation uses 3 register stacks. These allow the programmer to go across procedure calls without using the memory stack. Thus, we expect that our execution time should be quick.
- Short instruction syntax. Many of the instructions implicitly reference the stack and therefore have little to no operands leading to simple instructions for the programmer.

Registers

Since all the registers make up the stack, there are no registers directly available to the programmer. The only way for the programmer to access registers is to perform operations on the stack.

Procedure Call Conventions

When executing a procedure call, it is the caller's responsibility to place input arguments on the argument stack (a0 at top and a1 at next… etc) and it is the callee's responsibility to place its return value on the top of the return stack and not alter any of the arguments passed in on the procedure stack. The return addresses are stored on the return stack. The return addresses from the most recent procedure calls are at the top of the return stack. When returning from a

procedure, the return address can be accessed by the callee by popping the top of the return stack.

Addressing Modes

Instructions that implement jump use PC relative addressing, an example would be jnz. With PC relative addressing, the immediate is the instruction offset shifted left one then sign extended. This is added to the PC to determine the next instruction. For instructions that involve immediate values we use immediate addressing. An example would push or pop. This simply sign extends the immediate and stores it on the stack. Majority of Instructions that manipulate the stack utilize direct addressing. An example of this would be the swap instruction. The immediate field in this instruction is not used.

Instruction Format

Instruction Type	***15 12***	***11 6***	***5 4***	***3 0***
P-type	Imm[9:0]		opcode	
A-type	xxxxxxxxx		opcode	

Stack Visualization

Procedure Stack Visualization

arg1
arg2
…more args…
ra (return address)

Data Stack *Visualization*

Top
Next
Other data

Return Stack *Visualization*

RTop

RNext
Other ra's

ISA Table

Instruction Name	***Type***	***Opcode***	**Behavior**	***Description***
push	P	000010	DS[Top] = addr	Places a value from memory onto the top of the data stack
pushra	P	011111	RS[Top] = addr	Places a return address on the top of the return address stack
popra	P	101101	Addr = RS[Top] RS[Top] = next	Removes top from the register stack and places it in specific memory address.
pushi	P	100011	DS[Top] = imm[9:0]	Places an immediate onto the top of the data stack
pop	P	000011	Addr = DS[Top] DS[Top] = next	Removes Top from the data stack and places it in the specific memory address
peek	P	000101	Addr = DS[Top]	Saves top of data stack to specified memory address. (does not remove top)
jz	P	001101	Pop a	Pops top of data

			If a = 0, jump to label	stack. If top is equal to 0, then jump to label otherwise, fall through.
jnz	P	010011	Pop a If a != 0, jump to label	Pops top of data stack. If top is not equal to 0, then jump to label otherwise, fall through.
jmp	P	001110	pc = a	Unconditional jump
jal	P	111001	ra = pc + 4 pc = a	Jump and link instruction that jumps to address parameter and stores the address of the next instruction in ra
lup	P	010001	Push Big[15-6] Shift left 6	Loads upper 10 of large immediate
lli	P	010010	Push Big[5-0]	Loads lower 6 of large immediate
drop	A	111101	DS[Top] = DS[next]	Delete item at top of stack
sub	A	000110	DS[Top] = DS[nex]t - DS[Top]	Pops two items from data stack, subtract them, then put on top of data stack
add	A	000111	DS[Top] = DS[Next] + DS[Top]	Pops two items from data stack, add them, then put on top of data stack
Swap	A	001000	a = DS[Top] b = DS[Next] DS[Next] = a	Pops two items from the data stack, and

			DS[Top] = b	pushes them back to the data stack in the order they were popped.
Dup	A	001001	a = DS[Top] push a	Duplicates the item at the top of the data stack and pushes it to the data stack
eq	A	001010	DS[Top] = 0/1	Pushes a 1 to the top of the data stack if top and next are equal, 0 if not. (Doesn't pop anything)
lt	A	001011	DS[Top] = 0/1	Pushes a 1 to the top of the data stack if next < top. 0 Otherwise
le	A	010111	DS[Top] = 0/1	Pushes a 1 to the top of the data stack if next <= top. 0 otherwise
geq	A	001100	DS[Top] = 0/1	Pushes a 1 to the top of the data stack if next >= top. 0 otherwise
store	A	001111	PS[Top] = a	Pops the top of the data stack and pushes the value onto the procedure stack. This does remove the top from the data stack.
ret	A	010000	a = PS[Top]	Pops the top of the procedure

				stack and pushes the value into the top of the data stack. The does remove the item at the top of the procedure stack.
swapproc	A	111111	In procedure stack: a = PS[Top] b = PS[Next] PS[Next] = a PS[Top] = b	Swaps top and next in the procedure stack
dupproc	A	111110	a = PS[Top] push a to PS	Duplicates the top value in the procedure stack and pushes it back on

Memory Map

Stack(0XFFFF FFFF) Grows downwards Lower-bound: 0xFFFF FFFF Upperbound: 0xBBBB BBBB
Code(0X0000 4000) Grows upwards Upper-bound: 0xBBBB BBBB Lower-bound: 0x000 4000
Globals(0X0000 0000) Grows upwards Upper-bound: 0x0000 4000 Lower-bound 0x000 0000

Machine Language Translation

temp = 0x0000 0002
start = 0x0000 0004

Rel Prime Code to Assembly

Address	***RelPrime Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: pushi 2	***0000000010* 100011**	*// push 2 onto data stack*
0x4002	dup	***0000000000 001001***	*// duplicate the top of the stack (2 in this case)*
0x4004	store	***0000000000 001111***	*// store 2 to the procedure stack*
0x4006	swapproc	***0000000000 111111***	*// swap the top 2 elements in the procedure stack*
0x4008	GCD: dupproc	***0000000000 111110***	*// duplicate top item in procedure stack*
0x400A	ret	***0000000000* 010000**	*// retrieve item from top of procedure stack and push it into data stack*
0x400C	pushi 0	***0000000000* 100011**	*// push 0 onto the data stack*
0x400E	eq	**0000000000 001010**	*// pushes one onto data stack if top two items are equal*
0x4010	jz WHILE	**0000001000 001101**	*// jump 4 instructions (8 bytes) if a != 0*
0x4012	swap	***0000000000* 001000**	*// swap items at the top of data stack*
0x4014	store	***0000000000* 001111**	*// store top of data stack to top of procedure stack*
0x4016	jmp RELPRIME	***0000110110* 001110**	*// jump to RELPRIME, forward 27 instructions, or 54 bytes*
0x4018	WHILE: drop	***0000000000* 111101**	*//* delete value from

			top of data stack
0x401A	swapproc	***0000000000*** **111111**	*// stop top two items at top of procedure stack*
0x401C	dupproc	***0000000000*** **111110**	*// duplicate item at the top of procedure stack*
0x401E	ret	***0000000000*** **010000**	*// retrieve value from top of procedure stack and push into data stack*
0x4020	pushi 0	***0000000000*** **100011**	*// push 0 onto data stack*
0x4022	eq	***0000000000*** **001010**	*// push 1 onto stack if top two values are equal*
0x4024	jnz RETURN	***0000011110*** **010011**	*// jump to return if top of stack is not zero*
0x4026	drop	***0000000000*** **111101**	*// delete top value from data stack*
0x4028	le	***0000000000*** **010111**	*// push 1 onto data stack if second value is <= first value*
0x402A	jnz ANOTGREATER	***0000001010*** **010011**	*// got ANOTGREATER if top of stack is not 0*
0x402C	dup	***0000000000*** **001001**	*// duplicate top value of the data stack*
0x402E	pop temp	***0000000010*** **000011**	*// remove and store top value of stack to address temp*
0x4030	sub	***0000000000*** **000110**	*// subtract the top element of the data stack from the second element of the stack. Original values have been removed from stack.*

0x4032	push temp	***0000000010 000010***	*// push data at address temp to top of data stack*
0x4034	ANOTGREATER: swap	***0000000000*** **001000**	*// swap items at the top of data stack*
0x4036	dup	***0000000000*** **001001**	*// duplicate top value of the data stack*
0x4038	pop temp	***0000000010*** **000011**	*// remove and store top value of stack to address temp*
0x403A	sub	***0000000000*** **000110**	*// subtract the top element of the data stack from the second element of the stack. Original values have been removed from stack.*
0x403C	push temp	***0000000010 000010***	*// push data at address temp to top of data stack*
0x403E	swap	***0000000000*** **001000**	*// swap items at the top of data stack*
0x4040	jmp WHILE	***1111010100*** **001110**	*// jump to WHILE, back 22 instructions, or 44 bytes*
0x4042	RETURN: store	***0000000000*** **001111**	*// store top of data stack to top of procedure stack*
0x4044	jmp RELPRIME	***0000000010*** **001110**	*// jump to RELPRIME, 2 bytes forward.*
0x4046	RELPRIME: ret	***0000000000*** **010000**	*// retrieve value from top of procedure stack and push into data stack*
0x4048	pushi 1	***0000000001*** **100011**	*// push 1 onto the data stack*
0x404A	eq	**0000000000 001010**	*// push 1 onto stack if top two values are*

			equal
0x404C	jnz FINISH	***0000001110*** **010011**	*// got FINISH if top of stack is not 0*
0x404E	ret	***0000000000*** **010000**	*// retrieve value from top of procedure stack and push into data stack*
0x4050	pushi 1	***0000000001*** **100011**	*// push 1 onto the data stack*
0x4052	add	***0000000000*** **000111**	*// add the first two items on the stack together. These two items are removed from the stack.*
0x4054	store	***0000000000*** **001111**	*// store top of data stack to top of procedure stack*
0x4056	swap	***0000000000*** **001000**	*// swap items at the top of data stack*
0x4058	jmp GCD	***1110100100*** **001110**	*// jump to GCD (back 46 instructions, or 92 bytes)*
0x405A	FINISH: store	***0000000000*** **001111**	*// store top of data stack to top of procedure stack*
0x405C	ret	***0000000000*** **010000**	*//* transfer n to stack
0x405E	ret	***0000000000*** **010000**	*//* transfer m to stack
0x4060	ret	***0000000000*** **010000**	*// transfer*
0x4062	pop start	***0000000100*** **000011**	*// pop ra to save for jump*
0x4064	drop	***0000000000*** **111101**	*// get rid of m*
0x4066	push start	***0000000100*** **000010**	*// push ra back onto stack to be stored back into procedure stack*
0x4068	store	***0000000000*** **001111**	*// store ra back into*

			procedure stack
0x406A	store	***0000000000* 001111**	*// store n back into procedure stack*

Basic Operations

Assume:
b = 0x0000
c = 0x0002
a = 0x0004
i = 0x0006

Add two numbers:

High Level Code:
a = b + c;

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: push b	***0000000000 000010***	*// push value at address b onto data stack*
0x4002	push c	***0000000010 000010***	*// push value at address c onto the top of the data stack*
0x4004	add	***0000000000* 000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4006	pop a	***0000000100* 000011**	*// pop top of the data stack to address a*

High Level Code:
a = b + 5;

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: push b	***0000000000 000010***	*// push value at*

			address b onto data stack
0x4002	pushi 5	***0000000101*** **100011**	*// push the immediate 5 to the top of the data stack*
0x4004	add	***0000000000*** **000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4006	pop a	***0000000100*** **000011**	*// pop top of the data stack to address a*

Subtract two numbers:

High Level Code:

a = b - c;

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: push b	***0000000000 000010***	*// push value at address b onto data stack*
0x4002	push c	***0000000010 000010***	*// push value at address c onto the top of the data stack*
0x4004	sub	***0000000000*** **000110**	*// subtract the top element of the data stack from the second element of the stack. Original values have been removed from stack.*
0x4006	pop a	***0000000100*** **000011**	*// pop top of the data stack to address a*

High Level Code:

a = b - 5;

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: push b	***0000000000 000010***	*// push value at address b onto data stack*
0x4002	pushi 5	***0000000101*** **100011**	*// push the immediate 5 to the top of the data stack*
0x4004	sub	***0000000000*** **000110**	*// subtract the top element of the data stack from the second element of the stack. Original values have been removed from stack.*
0x4006	pop a	***0000000100*** **000011**	*// pop top of the data stack to address a*

Simple if statements:

High Level Code:

```
if (b == 0) {
b = 3 + b
}
a = b + c
```

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: pushi 0	***0000000000*** **100011**	*// push the immediate 0 to the top of the data stack*
0x4002	push b	***0000000000 000010***	*// push value at address b onto data stack*
0x4004	eq	***0000000000*** **001010**	*// push 1 onto the data stack if top two values are equal*
0x4006	jz SkipBody	***0000000100*** **001101**	*// pop top of the stack to address a*

0x4008	pushi 3	***0000000011* 100011**	*// push the immediate 3 to the top of the data stack*
0x400A	add	***0000000000* 000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x400C	pop b	***0000000000* 000011**	*// pop the top of the data stack to address b*
0x400E	SkipBody: push c	***0000000010 000010***	*// push value at address c onto the top of the data stack*
0x4010	add	***0000000000* 000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4012	pop a	***0000000100* 000011**	*// pop top of the data stack to address a*

<u>High Level Code:</u>
a = 3;
b = a + c;
if (a < b) {
// some code (not present in assembly)
}
a = b + c

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: pushi 3	***0000000011* 100011**	*// push the immediate 3 onto the top of the data stack*
0x4002	push c	***0000000010 000010***	*// push value at address c onto the top of the data stack*

0x4004	add	***0000000000* 000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4006	dup	***0000000000* 001001**	*// duplicate top value of the data stack*
0x4008	pop b	***0000000000* 000011**	*// pop the top of the data stack to address b*
0x400A	lt	***0000000000* 001011**	Pushes a 1 to the top of the data stack if next < top. 0 Otherwise
0x400C	jnz End	***0000001000* 010011**	*// got End if top of the data stack is not 0*
0x400E	push c	***0000000110 000010***	*// push value at address c onto the top of the data stack*
0x4010	add	***0000000000* 000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4012	pop a	***0000000100* 000011**	*// pop top of the data stack to address a*
0x4014	End:		

Loops:

<u>High Level Code:</u>

```
while (i > 0) {
        a = a + 1;
}
```

Address	***Assembly Code***	***Machine Code***	***Comments***
0x4000	PROGRAM: pushi 0	***0000000000* 100011**	*// push immediate 0 to the top of the data stack*

0x4002	push i	***0000000110 000010***	*// push value at address i onto the top of the data stack*
0x4004	While:		
0x4006	le	***0000000000*** **010111**	Pushes a 1 to the top of the data stack if next <= top. 0 Otherwise
0x4008	jnz End	***0000000100*** **010011**	*// goto End if top of the data stack is not 0*
0x400A	push a	***0000000100*** **000010**	*// push value at address c onto the top of the data stack*
0x400C	pushi 1	***0000000001*** **100011**	*// push immediate 1 to the top of the data stack*
0x400E	add	***0000000000*** **000111**	*// add the first two items on the data stack together. These two items are removed from the stack.*
0x4010	jmp While	***1111110100*** **001110**	*// jump to while unconditionally*
0x40142	End:		

Multicycle RTL

Push instructions	***Pop instructions***	***Jump instructions***	***Conditionals***	***Add/Sub***	**Drop**	**Dup Instructions**	**Swap Instructions**	**Store/ Ret**	**Peek**	**jal**
Pc <= Pc + 4 oldPc <= Pc Inst <= Mem[Pc]										
Stack.Nothing DTop <= DS[Top] DNext <= DS[Next] PTop <= PS[Top]										
Stack.shiftDown	Pop: Mem[instr[15:6]] = DS[Top] Popra: Mem[instr[15:6]] = RS[Top]	If (DS[Top] op 0) PC <= SE(instr[15:6]* 2)	Stack.shiftDown	DTop = DS[Top] DNext = DS[Next] Add: ALUOut = DNext + DTop Sub: ALUOut = DNext - DTop	Stack.shiftDown	Stack.shiftUp	Swap: DS[Top] = Dnext Dnext = DTop Swapproc: PS[Top] = Pnext Pnext = PTop	Store: PS.ShiftDown Ret: DS.ShiftDown	Mem[instr[15:6]] = DS[Top]	RS.ShiftDown
Push: Stack[Top] <= Mem[instr[15:6]] Pushra: Stack[Top] <= instr[15:6]	Stack.shiftUp		If (DTop op DNext) DS[Top] <= 1 Else DS[Top] <= 0	Stack.shiftDown		Dup: DS[Top] = DTop Dupproc: PS[Top] = PTop		Store: PS[Top] <= DTop Ret: DS[Top] <= PTop		RS[Top] <= inst[15:6]
				DS[Top] <= ALUOut						

Notes:

Stack.ShiftDown: This is a process that happens in the hardware due to a control signal. When this appears in the RTL, the hardware moves everything in the stack down one to open up the top of the stack

Stack.ShiftUp: This is a process that happens in the hardware due to a control signal. When this appears in the RTL, the hardware moves everything in the stack up one to ensure the top of the stack is not empty after a pop or add/sub.

Naming Conventions

Lower snake case will be used throughout.

Type	Convention	Example
Wires	Purpose of the wire is highlighted and starts with a ‘w’.	w_alu
Test Benches	Name of the module being tested prefixed with ‘tb’.	tb_control_unit
Modules	Purpose of the module is described	alu control_unit
Inputs	Purpose is described and prefixed with ‘in’	in_reset in_clk
Outputs	Purpose is described and prefixed with ‘ot’	ot_pc ot_data_out
Registers	Purpose is described and prefixed with ‘r’’	r_pc r_instruct
Clock	Purpose is described and prefixed with ‘clk’	clk_alu clk_main

Components

Component	Inputs/Control Signals	Outputs	Behavior	Implemented Components
Stack	Daction[1:0], input[9:0]	Top[15:0], Next[15:0]	The stack is an array of registers implemented with the LIFO approach.	Data stack, Procedure stack, Return stack
Register	Input_value[15:0], CLK[0:0], RST[0:0]	output[15:0]	On the rising edge of the clock (CLK) the register reads the input_value port and outputs the new value. If the reset signal is high on the rising clock edge the register wipes its data and outputs 0x0000 instead.	Dtop, Dnext, Ptop, Pnext, Rtop, Rnext, ALUut

ALU	**opA [15:0], opB [15:0], ALU_control [3:0]**	**ALU_result [15:0], zero [0:0]**	**The ALU performs arithmetic and logic operations based on the ALU_control input and opA and opB operands. It outputs the result of the operation and a zero flag that indicates if the result is zero.**	
Control Unit	**opcode [3:0]**	**ALU_control [3:0], mem_read [0:0], mem_write [0:0], branch [0:0], jump [0:0], Daction[0:1], Paction[0:1], Ration[0:1], Dinput[0:1],**	**The control unit takes in the opcode from the instruction register and generates control signals for the ALU, register write, memory read, memory write, branch, and jump operations.**	

		Pinput[0:1], Rinput[0:1], ALUsrcA[0:1], ALUsrcB[0:1]		
Immediate Generator	**Instruction[0:15],CLK [0:0]**	**Immediate of 16 bits which is sign extended**	**Handles the immediate arithmetic including sign extension.**	

Changes to Assembly and Machine Code Since M1

Changed to a 3-stack architecture. This third stack is called the return stack, and holds the return addresses for procedures.

Added instructions:

- pushra/popra - ability to push and pop off of return stack
- jal - allows ra to be pushed onto return stack to allow return from procedures