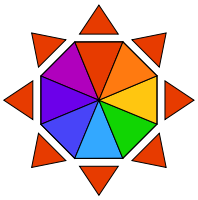
Jib Instruction Set Architecture

Ian O'Rourke

October 1, 2025



1 Overview

The Jib processor is a general-purpose Instruction Set Architecture (ISA). Each processor instruction is designed to take up exactly one word of memory. For Jib, each word takes up 32 bits; however, memory is addressable at the byte level. In cases where signed arithmetic is performed, signed integers are encoded using the two's-complement format. Jib also supports 32-bit floating point numeric algebra.

1.1 Registers

There are 32 total registers present on the Jib. These are defined generally as follows in Table 1. Note that, as Jib is a 32-bit architecture, this means that each register is 32 bits wide. While there are specifications between general-purpose and system-level registers, any register is addressable and usable via standard instructions equally.

Register	Type	Usage
R0	Reserved	Program Counter
R1	Reserved	Processor Status Flags
R2	Reserved	Stack Pointer
R3	Reserved	Load Offset
R4	Reserved	Return Register
R5	Reserved	Argument Base Register
R6	GP (C/Buoy)	Function Local Variables
R7	GP (C/Buoy)	Function Temporary Variables
R8	GP (C/Buoy)	Spare Register
R9-R31	GP	General Purpose Register

Table 1: Registers R0 through R5 are reserved for system-level parameters. R6 through R8 are reserved for C/Buoy functionality, by convention. All remaining registers are general-purpose.

The program counter indicates the next instruction to be read. At the beginning of the processor cycle, the instruction at the memory address of the program counter is read in and processed. Then, the program counter is incremented at the end of each instruction cycle. If this value is needed to be modified, it is recommended to use the absolute `jmp`, or the relative jump instruction `jmpr` (see Section 2), as opposed to writing to the register directly. This will automatically account for the increment at the end of the instruction cycle.

The global stack pointer maintains the global stack, as defined in Section 1.4. This provides the absolute address of the current stack location. Thus, when the stack is empty, it points to the stack base address, and when the stack is completely full it points to the memory location just above the last stack entry, or the base address plus the stack size. This value should not be edited by-hand to maintain the

consistency of the program execution, but is instead modified by the stack instructions `push`, `pop`, `popr`, `call`, `ret`, `retint`, and `int`, as well as hardware interrupts. This should be loaded by the init program by assembly, however, to provide the default base location for the stack.

The return value instruction is intended to store the result of a function call, made using the `call` instruction. When the processor status flags are replaced with the caller's flags after the `ret` instruction is called, the return value is the only register that remains unchanged.

The processor status flags indicate the current setup for the processor. Currently, the following flags are assigned, as noted in Table 2.

Bit	Value
0	Interrupt Enable
1	Interrupt Executing
2-7	Interrupt Number
8	Carry
...	...
31	...

Table 2: Processor status flags provide a window into the current processor state

This provides both a means to set and to read the current processor state values to ensure that the proper operating mode is configured for the currently-running program. This is maintained and replaced when `ret` and `retint` are called, so within an interrupt or function call, it is not necessary to replace the processor flags with those of the caller.

1.2 Overall Instruction Syntax

Since the architecture of the Jib is 32-bit, this means that the basic word, and therefore instruction format, exists in a 32-bit format. Note that, however, memory addresses are with respect to byte boundaries. Words are stored in big-endian format, meaning the most-significant component of the word is associated with the first byte in the word, while the least-significant component of the word is associated with the last byte in the word. Instructions must be placed with a base at a memory location with a byte alignment of four (e.g. 0, 4, 8, 12, etc.). General data may be placed and accessed from any byte alignment. The general instruction format is listed in Table 3. The first byte, Byte 0, is always split into two components - the ID, composing the first 4 bits of the instruction, and the SubID, composing the next 4 bits of the instruction.

	Byte 0	Byte 1	Byte 2	Byte 3
Location	0xFF000000	0x00FF0000	0x0000FF00	0x000000FF
Usage	opcode	arg0	arg1	arg2
Details	ID (`0xF0`), SubID (`0x0F`)	-	-	-

Table 3: The Jib instruction format typically has one opcode and three possible arguments associated with a particular opcode

This is contrasted with the typical assembly language formatting, which is provided as

```
instruction <arg2> <arg1> <arg0>
```

where the number of arguments depends on the required number of arguments for the instruction.

Certain arguments require type information. When this is present, the type code resides in the upper 3 bits of the argument byte (`0xE0`), while the register resides in the lower 5 bits (`0x1F`). Type codes are provided in Table 4.

1.3 Resetting

On a hard-reset, all registers will be reset to 0 and memory values will be reset to their default values. The data parameter in memory at the hard-reset vector, stored at memory location 0, will be used as the reset vector.

Type	Code	Size in Bytes	Description
U8	1	1	Unsigned byte
I8	2	1	Signed byte
U16	3	2	Unsigned short
I16	4	2	Signed short
U32	5	4	Unsigned integer
I32	6	4	Signed integer
F32	7	4	32-bit floating point

Table 4: The supported type codes that are associated with certain instruction codes

On soft-reset, as called by the reset the program counter will be assigned the value contained in the soft-reset vector, which is stored at memory location 1. All of the other registers are reset to their default value of 0. Processor memory will be left unchanged and processor execution is then started.

On any reset, interrupts will be disabled by default, and must be enabled manually.

1.4 The Stack

The stack pointer provides the absolute address of the stack pointer. The pointer points to the memory location just above the current stack location. If the stack is empty, the stack pointer points to the base address. Note that the base address is user-selectable, and there are no protections for stack under or overflow conditions, outside of wrapping around the minimum or maximum memory address, where the processor will error and halt.

1.5 Interrupts

Interrupts provide a means to interrupt the current flow of execution and run a separate method. There are two types of interrupts - hardware interrupts, which originate by request of an external hardware device, and software interrupts, which originate from a specific instruction. When an interrupt is triggered, the flow of program execution is interrupted before the next instruction is started. The current register state is stored on the stack, and the program counter is replaced with the value in the corresponding interrupt vector. The status register is also updated with the current interrupt execution data. Then, the program execution continues from this new location.

The Jib supports 32 software and 32 hardware interrupts, as shown in Table 5. The hardware interrupt vectors are stored in memory at locations `0x100` for interrupt 0 though `0x180`, exclusive, for interrupt 31. Similarly, the software interrupts are stored in memory at memory addresses `0x180` for interrupt 0 through `0x200`, exclusive, for interrupt 31.

Type	0	1	2	...	29	30	31
Hardware	`0x100`	`0x104`	`0x108`	...	`0x174`	`0x178`	`0x17C`
Software	`0x180`	`0x184`	`0x188`	...	`0x1F4`	`0x1F8`	`0x1FC`

Table 5: Interrupt vector locations for both hardware and software interrupts

Interrupts are processed by the interrupt controller. This stores all interrupt requests and provides the CPU the next interrupt to process based on their priority. Note that, if a vector has a value of zero, that interrupt vector is considered to be disabled and that interrupt will effectively be disabled and not able to be run.Once interrupts are re-enabled, if this queue is not empty, then that interrupt will be run, in priority-order.

At the conclusion of any interrupt, the `retint` instruction should be called. This will replace the current register state with the values provided off the stack and resume program execution from the location right after the interrupt was called. It will also clear the It should be noted that, if any values were pushed onto the stack during the interrupt handler, they should be popped off the stack prior to the `retint` call to avoid corrupting the program state. The `retint` instruction will return from the interrupt handler and allow the CPU to mark this interrupt as completed, so that it can move on to the next interrupt handler, or stop interrupt handling and resume normal operation if there are no more interrupts to be run. By clearing the interrupt execution status at the end of the handler, this prevents duplicates of the same interrupt from being run, preventing a execution errors if a hardware device spams the associated hardware interrupt.

2 Instructions and Assembly Code

All available instructions are listed in Table 6. Note that any invalid instruction that is not provided in the table below results in an immediate halt of the processor. Note that, in the below logic, any indication where PC is incremented indicates that the standard `PC += 1` to move to the next instruction will be replaced by the logic provided in the description field. Note that, based on the type, for addresses, this can affect either just the memory location assigned by the register (for a single-byte type), the register byte and the following byte (for a two-byte type), or the register byte and the following three bytes (for a four-byte type). The user must ensure that the appropriate locations are valid and able to be written to when setting up the registers and types for particular instructions. Instruction formats, as denoted in the "Format ID" column, are located in Table 7.

Format ID	Opcode		Assembly	Description
	ID	SubID		
A	0	0	`noop`	No Operation
A	0	1	`reset`	`$PC = Reset Vector`, `R[0-15] = 0`
B	0	2	`int <imm>`	Trigger software interrupt number `imm`
C	0	3	`intr [a]`	Trigger software interrupt number `R[a]`
A	0	4	`retint`	$\forall_{i \in [31 \rightarrow 0]}$ `R[i] = mem[--$SP], ++$PC`
C	0	5	`call [a]`	$\forall_{i \in [0 \rightarrow 31]}$ `mem[$SP++] = R[i]; $PC = R[a]`
A	0	6	`ret`	$\forall_{i \in [31 \rightarrow 0], i \neq \$ret}$ `R[i] = mem[--$SP], ++$PC`
C	0	7	`push [a]`	`mem[$SP += 4] = R[a]`
A	0	8	`pop`	`$SP -= 4`
C	0	9	`popr [a]`	`R[a] = mem[$SP -= 4]`
C	0	10	`jmp [a]`	`$PC = R[a]`
C	0	11	`jmpr [a]`	`$PC += R[a]`
B	0	12	`jmpri <imm>`	`$PC += Imm` (Signed)
A	0	15	`halt`	Stop Program Execution
G	1	0	`ld [a] [b]`	`R[a] = mem[R[b]]`
G	1	1	`ldr [a] [b]`	`R[a] = mem[$PC + R[b]]`
E	1	2	`ldi [dst] <imm>`	`R[dst] = Im`
E	1	3	`ldri [dst] <imm>`	`R[dst] = mem[$PC + Im]`
D	1	4	`ldn [a]`	`R[a] = mem[$PC + 4], $PC += 8`
D	1	5	`ldno [a]`	`R[a] = mem[$LDO + PC + 4], $PC += 8`
G	1	6	`sav [a] [b]`	`mem[R[a]] = R[b]`
G	1	7	`savr [a] [b]`	`mem[$PC + R[a]] = R[b]`
F	1	8	`copy [a] [b]`	`R[a] = R[b]`
H	1	9	`conv [a] [b]`	`R[a] = R[b]`
I	2	0	`teq [dst] [a] [b]`	If `R[a] == R[b] R[dst] = 1`, Else `R[dst] = 0`
I	2	1	`tneq [dst] [a] [b]`	If `R[a] != R[b] R[dst] = 1`, Else `R[dst] = 0`
I	2	2	`tg [dst] [a] [b]`	If `R[a] > R[b] R[dst] = 1`, Else `R[dst] = 0`
I	2	3	`tge [dst] [a] [b]`	If `R[a] >= R[b] R[dst] = 1`, Else `R[dst] = 0`
I	2	4	`tl [dst] [a] [b]`	If `R[a] < R[b] R[dst] = 1`, Else `R[dst] = 0`
I	2	5	`tle [dst] [a] [b]`	If `R[a] <= R[b] R[dst] = 1`, Else `R[dst] = 0`
F	3	0	`not [dst] [a]`	If `R[a] != 0 R[dst] = 0`, Else `R[dst] = 1`
F	3	1	`bool [dst] [a]`	If `R[a] != 0 R[dst] = 1`, Else `R[dst] = 0`
C	3	2	`tz [a]`	If `R[a] == 0 PC += 1`, Else `PC += 2`
C	3	3	`tnz [a]`	If `R[a] != 0 PC += 1`, Else `PC += 2`
A	4	0	`inton`	Turn Interrupts On, set `$STAT`
A	4	1	`intoff`	Turn Interrupts Off, set `$STAT`
A	5	0	`brk`	Debug Breakpoint (capturable by hardware; otherwise, acts as a `noop`)
I	10	0	`add [dst] [a] [b]`	`R[dst] = R[a] + R[b]`
I	10	1	`sub [dst] [a] [b]`	`R[dst] = R[a] - R[b]`
I	10	2	`mul [dst] [a] [b]`	`R[dst] = R[a] * R[b]`
I	10	3	`div [dst] [a] [b]`	`R[dst] = R[a] / R[b]`
I	10	4	`rem [dst] [a] [b]`	`R[dst] = R[a] % R[b]`
G	10	5	`neg [dst] [a]`	`R[dst] = -R[a]`
I	11	0	`band [dst] [a] [b]`	`R[dst] = R[a] & R[b]`
I	11	1	`bor [dst] [a] [b]`	`R[dst] = R[a] | R[b]`
I	11	2	`bxor [dst] [a] [b]`	`R[dst] = R[a]` $\wedge$ `R[b]`
I	11	3	`bshl [dst] [a] [b]`	`R[dst] = R[a] << R[b]`
I	11	4	`bshr [dst] [a] [b]`	`R[dst] = R[a] >> R[b]`
G	11	5	`bnot [dst] [a]`	`R[dst] = ~R[a]`

Table 6: Available instruction list for the Jib provides a variety of commands.

There are several different types of instruction formats, which make use of the four byte spots in the instruction opcode slightly differently. These are provided below in Table 7. As the first byte, Byte 0, is always the same, it is omitted from the table below. See Table 3 for more details on Byte 0. Most

commonly, and instruction will only use a single type for all registers present in the operation. Some instructions, in particular conversion instructions, will use multiple type codes for different registers.

ID	Name	Byte 1	Byte 2	Byte 3
A	No Argument	-	-	-
B	Immediate (i16)	-	Imm (`0xF0`)	Imm (`0x0F`)
C	Single Register	Register	-	-
D	Single Register Type	Register Type	-	-
E	Single Register Immediate	Register Type	Imm (`0xF0`)	Imm (`0x0F`)
F	Double Register	Register	Register	-
G	Double Register Type	Register Type	Register	-
H	Conversion	Register Type	Register Type	-
I	Arithmetic	Register Type	Register	Register

Table 7: Instructions can take a variety of different forms depending on the needs of the operation

The assembler also has several commands available, detailed in Table 8. Note that, for the `.loadtext` command, the text will be loaded in via the character map provided in Section 3.

Command	Description
`:[label]`	Defines a new label associated with the current memory location
`.oper [offset]`	Changes the current assembly offset to the value provided
`.load [num]`	Loads the data value as either an unsigned word (if in hex or positive) or as a signed word (if negative) in the current memory location
`.loadloc [label]`	Loads the data index associated with the provided label into the current memory location
`.text "[TEXT]"`	Loads the text into memory, starting at the current memory location, placing each character into the next subsequent memory location, with a null-terminator as copied into memory after the text value
`.reserve [num]`	Reserves the the next `[num]` bytes of memory for data, filling the memory with zeros
`.u8/.u16/.u32 [num]`	Loads the given unsigned integer at the current location
`.i8/.i16/.ui32 [num]`	Loads the given signed integer at the current location
`.f32 [num]`	Loads the given floating point number at the current location

Table 8: Available assembler commands

Reference names are available to link to the special register values, as listed in Table 9.

Reference Name	Register	Description
`$pc`	0	Program Counter
`$stat`	1	Status Flags
`$sp`	2	Stack Pointer
`$ldo`	3	Load Offset
`$ret`	4	Return
`$arg`	5	Argument Base

Table 9: Assembler provides shortcuts for commonly-referenced register indices

2.1 Calling Convention

The calling convention allocates the required arguments in a contiguous block of memory, one after another. The address of the start is placed in the argument base register, and then the function may be called. For return variables, primitives should be placed directly in the output register. If a larger value is desired to be output than may be placed directly in the return register, then the desired output address should be created by the caller and assigned to the return register prior the function call. The function may then be called, where it will write to the provided address in the return register. In this case, the return register should not be directly written to, but instead used as a if it were a pointer argument. This

convention should be followed for both assembly functions as well as higher-level language functions, if possible, for compatibility between the two.

3 Character Mapping

Jib computers, by default, utilize the following character map. This is similar to the American Standard Code for Information Interchange (ASCII) format. The Jib character map is defined in Table 10. Any undefined entries in the character map are considered invalid characters.

Hex	Char	Hex	Char	Hex	Char	Hex	Char
00	\0 NULL	20	Space	40	@	60	‘
01		21	!	41	A	61	a
02		22	"	42	B	62	b
03		23	#	43	C	63	c
04		24	$	44	D	64	d
05		25	%	45	E	65	e
06		26	&	46	F	66	f
07		27	'	47	G	67	g
08		28	(	48	H	68	h
09		29	)	49	I	69	i
0A	\n New Line	2A	*	4A	J	6A	j
0B		2B	+	4B	K	6B	k
0C		2C	,	4C	L	6C	l
0D		2D	-	4D	M	6D	m
0E		2E	.	4E	N	6E	n
0F		2F	/	4F	O	6F	o
10		30	0	50	P	70	p
11		31	1	51	Q	71	q
12		32	2	52	R	72	r
13		33	3	53	S	73	s
14		34	4	54	T	74	t
15		35	5	55	U	75	u
16		36	6	56	V	76	v
17		37	7	57	W	77	w
18		38	8	58	X	78	x
19		39	9	59	Y	79	y
1A		3A	:	5A	Z	7A	z
1B		3B	;	5B	[	7B	{
1C		3C	<	5C	\	7C	|
1D		3D	=	5D	]	7D	}
1E		3E	>	5E	^	7E	~
1F		3F	?	5F	_	7F	

Table 10: Jib character mapping

4 Devices

Devices are memory-mapped in Jib. This means that reading or writing to special regions in memory facilitate the communication with these external devices. In a typical Jib computer, the device region consists of up to 64 devices, starting at memory address `0xA000`, with each device allocating up to 32 bytes of memory. Not all devices will make use all the available memory slots for a given device. In these cases, a memory exception will be provided if any of these invalid addresses are read from or written to.

4.1 Serial Input and Output

The serial input and output device is one of the simplest devices. It essentially consists of a device two queues, one for input, and another for output. Each of these queues has an internal buffer size of 256 words. If any words are attempted to be added to the queue once either queue is full, no additional data is read and that data is lost.

The memory mapping for the serial input and output device is provided in Table 11.

Offset	Type	Read/Write	Usage
`0`	u16	Read	Device ID 1
`2`	u8	Read	Provides the current input queue size
`3`	u8	Read	Pops and provides a word off the front of the input queue If no data is available, will return `0` by default
`4`	u8	Read	Provides the current output queue size
`5`	u8	Write	Pushes a word onto the output queue
`6`	u8	Write	Clears the input queue if the value written is nonzero
`7`	u8	Write	Clears the output queue if the value written is nonzero
`8`	u8	Read/Write	If non-zero, interrupt to write to when input is received

Table 11: Serial Input and Output device provides a simple data structure to read and write data streams

4.2 IRQ Clock

The IRQ clock provides a means to trigger a specific hardware interrupt at a regular interval of clock cycles. This consists of a settable interval (or set to 0 to disable), as well as a settable interrupt to trigger. Available in memory is the ability to read any of these two settable parameters, as well as a readable indication of the current clock cycle count. The memory mapping is provided in Table 12.

Offset	Type	Read/Write	Usage
`0`	u16	Read	Device ID 2
`2`	u32	Read/Write	Gets/sets the clock interval in CPU cycles. Set to 0 to disable.
`6`	u32	Read	Provides the current clock counter in CPU cycles. Will be between 0 and the above interval.
`10`	u32	Read/Write	The hardware IRQ to trigger, if non-zero.

Table 12: Serial Input and Output device provides a simple data structure to read and write data streams

5 Examples

The following list some simple example programs that can be run on the Jib.

5.1 Counter

The program listed in Listing 1 provides a basic counter. A target value is placed in register four, and the value in register three is incremented from 0 to the target value in register four by adding one to the register each loop. Once the target value has been reached and register three is equal to register four, the program halts by entering an infinite loop.

```
;; Counter Program

; Define the hard-reset vector location
.loadloc start

; Define the soft-reset vector location
.loadloc start

.oper 0x2000
:functest
ldi $ret:i16 97
ret

; Define the starting location
.oper 0x4000
:start
ldn $sp:u16
.u16 0x1000

; Setup the counter variables
ldi 6:i16 1
ldi 7:i16 0
ldi 13:i16 0
ldi 15:i16 100

; Setup the target value
jmpri load_data

:target_value
.u16 64 ; 0x7FFF

.align
:load_data
ldri 8:u16 target_value

; Load the function call test
ldri 15:u32 functest_loc
jmpri post_load
:functest_loc
.loadloc functest
:post_load
call 15
copy $ret 12

; Perform the addition and check for reaching the target
:loop
add 7:i32 6 7
int 0
sub 9:i32 8 7

; Jump back to the add instruction if the
; target minus current is greater than zero
tl 10:i32 7 8
tz 10
jmpri endloc
jmpri loop

; Otherwise, enter an infinite loop as program completion
:endloc
jmpri endloc
```

Listing 1: Limited counter program

5.2 Infinite Counter

Listing 2 makes use of some additional functionality, including the use of the register reference names, to test register reset functionality. It also modifies the program counter outside of the standard jump command.

```
;; Infinite Counter Program

; Define the starting location
.loadloc start

; Move to the starting location
.oper 0x2000
:start
ldn $sp:u32
.u32 0x1000

; Load initial values
ldi 6:u16 1
ldi 7:u16 0

:loop ; define the main loop
add 7:u32 7 6
jmpri loop
```

Listing 2: Infinite counter program with register reference names

5.3 Hello World

Listing 3 provides a program that will infinitely write "hello, world" to the serial device, expected to be memory-mapped into the Jib at `0xA000`. This makes use of the `.loadtext` command, which loads the text, and an additional null-terminator at the end, directly into memory. It also makes use of a print-string function call to handle printing string values to the serial output devices.

```
;; Hello World Program

; Define the hard-reset vector location
.loadloc start

; Define the soft-reset vector location
.loadloc start

; Define the starting location
.oper 0x1000
:start
ldn $sp:u16
.u16 0x2000

; Load the string location into memory
ldn 14:u32
.loadloc str_hello_world

; Load the function location into memory
ldn 15:u32
.loadloc func_print_str

:loop
copy $arg 14
call 15
jmpri loop

.oper 0x1100
:str_hello_world
.text "hello, world!"

.oper 0x1200
:func_print_str
    ldn 13:u16
    .u16 0xA000

    ; Mark the location to write serial values to
    ldi 15:u16 5
    add 15:u32 15 13

    ; Load the argument value
    copy 6 $arg
    ldi 8:u16 1

    :func_print_str_loop
        ld 7:u8 6
        tz 7
            jmpri func_print_str_end
            sav 15:u8 7
        add 6:u32 8 6
        jmpri func_print_str_loop

    :func_print_str_end
    ldi 7:u16 10
    sav 15:u8 7
    ret
```

Listing 3: Infinite "hello, world!" program

5.4 Serial Echo

Listing 4 provides a serial echo program, which echos any text provided by the default serial input device back out via the serial echo device. This serial device is expected to be provided at the starting location `0xA000`.

```
;; Serial Echoer

; Define the hard-reset vector location
.loadloc program_start

; Define the soft-reset vector location
.loadloc program_start

.oper 0x4000
:program_start
ldn $sp:u32
.u32 0x1000
ldn 13:u32
.u32 0xA000

; Mark the location to read input values from
ldi 14:u16 3
add 14:u32 14 13

; Mark the location to write serial values from
ldi 15:u16 5
add 15:u32 15 13

; Mark the location to check the queue size from
ldi 12:u16 2
add 12:u32 12 13

:main_loop
ld 6:u8 12
tz 6
jmpri main_loop_end

ld 7:u8 14
sav 15:u8 7

jmpri main_loop

:main_loop_end
jmpri main_loop
```

Listing 4: Serial echo program reads in text characters and immediately outputs via the output device

6 C/Buoy

C/Buoy is a higher-level language than the raw assembly that provides some nicer tools to program applications in, at the cost of being less space and time efficient than programming in assembly directly. It does offer a slightly safer method of computing, however, as it provides a more traditional function call structure, variables, and scopes that help abstract away some of the hardware and provide some simple limits on what the user is allowed to do.

Note that, in this case, both functions and variables share the same namespace. This allows for easy use of function pointers by variable names, though currently only names are allowed, and no expressions may be used as yet.

Program	→
	⟨BaseStatement⟩
BaseStatement	→
	⟨BaseStatement⟩⟨BaseStatement⟩
	fn `FunctionName`(`Variable`[, `Variable`...]) { ⟨StatementList⟩}
	asmfn `FunctionName`(`Variable`[, `Variable`...) { [⟨WordLiteral; ⟩...] }
	global VarType `Variable`[= ⟨BaseExpression⟩];
Statement	→
	def `Variable`: ⟨VarType⟩[= ⟨BaseExpression⟩];
	return;
	return ⟨BaseExpression⟩;
	⟨BaseExpression⟩;
	{ ⟨StatementList⟩}
	if (⟨BaseExpression⟩) ⟨Statement⟩
	if (⟨BaseExpression⟩) ⟨Statement⟩else ⟨Statement⟩
	while (⟨BaseExpression⟩) ⟨Statement⟩
StatementList	→
	⟨Statement⟩
	⟨Statement⟩⟨Statement⟩
BaseExpression	→
	⟨Expression⟩= ⟨BaseExpression⟩
	⟨Expression⟩⟨BinaryOp⟩⟨Expression⟩
	⟨Expression⟩
Expression	→
	`Variable`
	⟨Literal⟩
	⟨UnaryOp⟩⟨Expression⟩
	`FunctionName`([⟨BaseExpression⟩[, ⟨BaseExpression⟩,...]])
	(⟨BaseExpression⟩)
	⟨Expression ⟩[⟨Expression ⟩]
BinaryOp	→
	+, -, *, /, <, >, <=, >=, &&, ||, &, |, ==, !=
UnaryOp	→
	*, -, +, !, &, ~
VarType	→
	⟨PrimitiveType ⟩
	*⟨VarType ⟩
	[⟨NumericValue ⟩]⟨VarType ⟩

This makes use of a few extra conventions. At the start of each function, stack space is reserved both for any local variables, as well as any temporary variables, that are required. A few extra registers are defined in the Table 1.

When returning a parameter, if the parameter is a primitive, it is returned by value in the return register directly. However, if it is a structure, the address of the temporary to write into is written into the return register before the function call. This is typically a temporary allocated on the stack, as mentioned above.

A simple preprocessor allows for mixing multiple files into one source unit. This follows a similar

syntax to the C preprocessor, albeit much simpler.

- `#include <file>` includes the provided file into the current source.
- `#ifdef <name>` and `#ifndef <name>` only includes the subsequent lines if the name is defined or not defined. Must have a matching `#endif`.
- `#else` is coupled with an `#if*` statement to include if the original statement is false. Only one may be provided per `#if*` statement.
- `#endif` ends an if statement.
- `#define <name>` marks the provided name as defined.
- `#undef <name>` marks the provided name as undefined.

6.1 A Simple Example

This provides an example program in Listing 5 below, which provides an example of printing text to a serial device, both using a string literal, but also converting numeric values into their text equivalents.

```
global TEST_STR: *u8 = "Hello, world!";
global SERIAL_OUT_LOC: *u8 = 0xA000u32 + 5;

fn print(c: *u8) void {
    while (*c != 0) {
        *SERIAL_OUT_LOC = *c;
        c = c + 1;
    }
}

fn print_digit(v: u8) void {
    if (v < 10) {
        *SERIAL_OUT_LOC = '0' + v;
    } else {
        *SERIAL_OUT_LOC = '?';
    }
}

fn print_uint(v: u32) void {
    def chars: [12]u8;
    def cp: *u8 = &chars;
    if (v == 0) {
        print_digit(0);
    } else {
        while (v != 0) {
            *cp = v % 10;
            v = v / 10;
            cp = cp + 1;
        }
        while (cp != &chars[0]) {
            cp = cp - 1;
            print_digit(*cp);
        }
    }
}

fn main() void {
    print_uint(s.a);
    print("\n");
    print_uint(s.b);
    print("\n");
    print_uint(1234);
    print("\n");
    print(TEST_STR);
    print("\n");
}
```

Listing 5: Example C/Buoy program for printing out a string to a serial output device

C/Buoy allows working with structs, primitives, functions, and more. This allows for an easier programming experience, at the cost of occasionally some additional generated machine code, resulting in slightly larger execution times, than could necessarily be done by hand in Jib assembly.

6.2 Assembly Integration

To allow for additional optimization, Jib assembly code can also be mixed in with functions to increase the speed of certain functions. As the compiler is currently not an optimizing compiler, this provides the opportunity to convert commonly-used smaller functions into compact assembly code to increase the speed of execution. Identifiers from the overall C/Buoy program can be mixed into the assembly code by using replacement characters, as denoted in Table 13.

For example, the above listing could have the print function replaced by the print function in List 6 and be considerably more efficient.

Parameter	Format	Result
Global Variable / Function Location	`%{IDENTIFIER}%`	Provides the entry/access label for the provided function or global variable
Global Literal Value	`^{IDENTIFIER}^`	Provides the raw value of the provided literal/constant at the global scope
Local Variable Offset	`@{IDENTIFIER}@`	Provides the offset (from the argument base) for the given local variable or function parameter
Member Offset	`&{IDENTIFIER}&`	Provides the offset of the provided structure member value
`sizeof`	`#{IDENTIFIER}#`	Provides the `sizeof` for the provided type or symbol

Table 13: Formats for parameters that are replaced in the assembly code with the resulting value for better interoperability with the overall function

```
#ifndef K_SERIAL_IO
#define K_SERIAL_IO

global SERIAL_IN_SIZE: *u8 = 0xA000u32 + 2;
global SERIAL_IN_LOC: *u8 = 0xA000u32 + 3;
global SERIAL_OUT_LOC: *u8 = 0xA000u32 + 5;

#ifdef K_SERIAL_IO_CBUOY_PRINT
fn print(c: *u8) void {
    while (*c != 0) {
        *SERIAL_OUT_LOC = *c;
        c = c + 1;
    }
}
#else
asmfn print(c: *u8) void {
    "push $stat";
    "intoff";
    "ldn 8:u32";
    ".loadloc %{SERIAL_OUT_LOC}%";
    "ld 8:u32 8";
    ".align";
    "ldi 9:u16 1";
    "ld 10:u32 $arg";
    "ldi 11:u16 @{c}@";
    "add 10:u32 10 11";
    "ld 11:u8 10";
    "tz 11";
    "jmpri 16";
    "sav 8:u8 11";
    "add 10:u32 10 9";
    "jmpri -20";
    "popr $stat";
    "ret";
}
#endif

fn print_digit(v: u8) void {
    if (v < 10) {
        *SERIAL_OUT_LOC = '0' + v;
    } else {
        *SERIAL_OUT_LOC = '?';
    }
}

fn print_uint(v: u32) void {
    def chars: [12]u8;
    def cp: *u8 = &chars;
    if (v == 0) {
        print_digit(0);
    } else {
        while (v != 0) {
            *cp = v % 10;
            v = v / 10;
            cp = cp + 1;
        }
        while (cp != &chars[0]) {
            cp = cp - 1;
            print_digit(*cp);
        }
    }
}

fn print_stat(name: *u8, v: u32) void {
    print(name);
    print(": ");
    print_uint(v);
    print("\n");
}

#endif // K_SERIAL_IO
```

Listing 6: Replacement print function to improve the speed of the resulting output

The original program in Listing 5 runs to a halt in 1320 CPU cycles. After replacing the print function with the one in Listing 6, the same program completes to a halt in 1010 CPU cycles. Used appropriately, similar optimizations can be made throughout program code to target and improve execution time for different functions as-needed. This also the programmer to slowly learn assembly language by starting

with a higher-level programming language as an entry-point, and then working to the lower-level functionality gradually over time as they get familiar with the system. Assembly language is required for several key pieces of functionality, including setting up things to manage the stack directly - such as creating threads - and for interfacing more efficiently with hardware input/output devices.

6.3 Threading

Through direct modification of the stack, we can create a set of utilities to swap between different threads at runtime. Listing 7 provides several functions. `k_tsk_init` provides means to register a new thread. `k_tsk_main` is a function that is called to transition from the main thread into the inner executing threads. `k_tsk_yield` is both an interrupt handler (for a periodic device like a timer) and also a function that may be called by threaded functions to yield their execution to the next thread. This is simple round-robin threading, so there is no scheduler implemented in this example.

```
#ifndef K_TSK
#define K_TSK

#include serialio.cb

const K_TSK_STACK_SIZE: u32 = 1024;
const K_TSK_STACK_NUM: u32 = 4;
global K_TSK_THREAD_NUM: u32 = 0;
global K_TSK_THREAD_CURR: u32 = 0;

struct k_tsk_thread_stack_t {
    last_ptr: u32;
    stack: [K_TSK_STACK_SIZE]u8;
}

global K_TSK_THREAD_STACK: [K_TSK_STACK_NUM]k_tsk_thread_stack_t;

asmfn k_tsk_main() void {
    "ldn 8:u32";
    ".loadloc %{K_TSK_THREAD_NUM}%";
    "ld 8:u32 8";
    "tz 8";
    "ret";
    "ldn 9:u32";
    ".loadloc %{K_TSK_THREAD_STACK}%";
    "ld 9:u32 9";
    "copy $sp 9";
    "ret";
}

asmfn k_tsk_yield() void {
    "intoff";
    "; Increment K_TSK_THREAD_NUM to next thread";
    "ldn 8:u32";
    ".loadloc %{K_TSK_THREAD_NUM}%";
    "ld 8:u32 8";
    "ldn 9:u32";
    ".loadloc %{K_TSK_THREAD_CURR}%";
    "ld 10:u32 9";
    "copy 16 10";
    "ldi 11:u16 1";
    "add 10:u32 10 11";
    "rem 10:u32 10 8";
    "sav 9:u32 10";
    "; Swap stack pointer";
    "; 10 = new pointer";
    "; 16 = old pointer";
    "ldi 8:u16 #{k_tsk_thread_stack_t}#";
    "mul 10:u32 10 8";
    "mul 16:u32 16 8";
    "ldn 9:u32";
    ".loadloc %{K_TSK_THREAD_STACK}%";
    "add 10:u32 10 9";
    "add 16:u32 16 9";
    "sav 16:u32 $sp";
    "ld $sp:u32 10";
    "inton";
    "retint";
}

fn k_tsk_init(func: fn() void) void {
    if (K_TSK_THREAD_NUM < K_TSK_STACK_NUM) {
        def current: *k_tsk_thread_stack_t = &K_TSK_THREAD_STACK[K_TSK_THREAD_NUM];
        def current_stack: *u32 = &((*current).stack);
        print_stat("Base Struct", current);
        print_stat("Base Stack", current_stack);
        current_stack[0] = func;
        current_stack[1] = 1; // Enable Interrupts on the new stack
        print_stat("Function Location", func : u32);
        current_stack[2] = (current_stack + 32) : u32;
        current->last_ptr = current_stack[2];
        print_stat("Stack Location", (*current).last_ptr);
        print_stat("Stack Size", (((*current).last_ptr) : u32) - (current_stack : u32));
        K_TSK_THREAD_NUM = K_TSK_THREAD_NUM + 1;
        print_stat("Current Thread Count", K_TSK_THREAD_NUM);
        print("\n");
    } else {
        print("Exceed allowed thread count\n");
    }
}

#endif // K_TSK
```

Listing 7: Utilities that use a mix of Jib assembly and C/Buoy provide mechanisms to create threads at runtime and swich between threads, either through preemptive interrupts or task yielding.

7 Tools

Several tools can help in the development of Jib programs.

7.1 V/Jib

One useful tool is `V/Jib`, combines together a basic assembler, CPU emulator, and memory inspector into a single program. The main window can be seen in Figure 1.

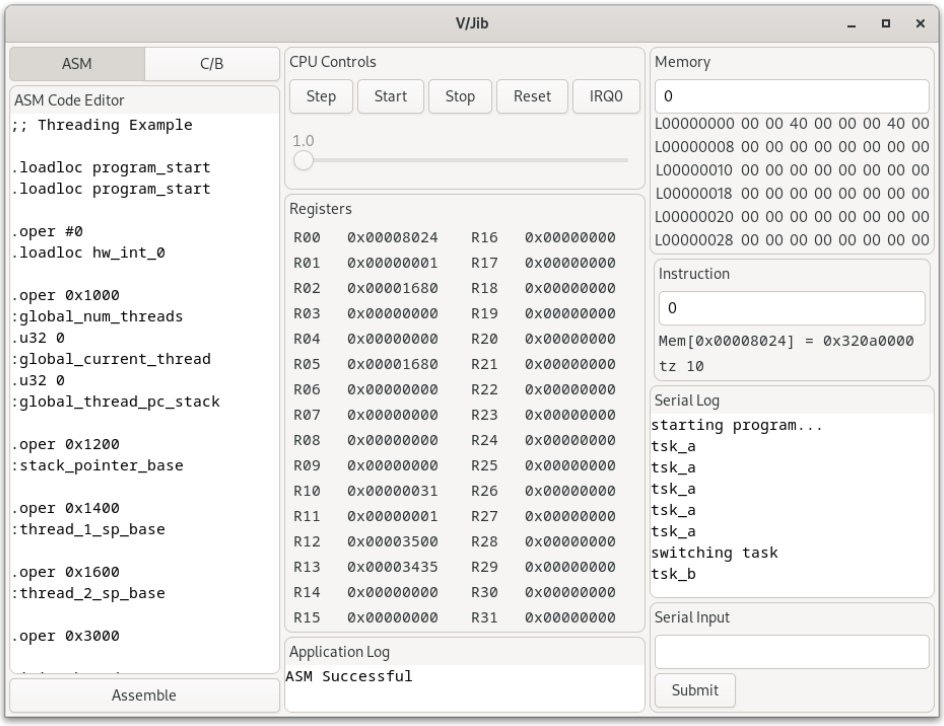


Figure 1: Main window of `V/Jib` provides common tools for program writing