PQR5 Assembler

Instruction Manual

Aug 2024

Contents

1. General Info

pqr5asm is an assembler which translates RISC-V assembly to binary/hex code.

Compliance	RV32I (User-Level ISA v2.2) - 37 base instructions + pseudo/custom instructions
Input	Assembly program with .s extension
Output	Binary/Hex code for program & data in ASCII text and .bin formats: • sample_imem.bin – Program binary • sample_imem_bin.txt – Binary text file of program, human readable • sample_imem_hex.txt – Hex text file of program, human readable • sample_dmem.bin – Data binary • sample_dmem_bin.txt – Binary text file of data, human readable • sample_dmem_hex.txt – Hex text file of data, human readable

Syntax Rules	1) One instruction per line.
	2) Every program requires the program section, `.section .text`, and base address of the program should be defined by linker directive, `.org` for eg:
	`.section .text`
	`.org 0x00000000`
	3) Supports <space>, <comma>, and <linebreak> as delimiters for eg:
	`LUI x5 255` <linebreak> or `LUI x5, 255` <linebreak>
	4) Use '#' for inline/newline comments for eg:
	`LUI x5, 255  # This is a sample comment`
	5) Supports 32-bit signed/unsigned integer, 0x hex literals for immediate.
	For eg: `255, 0xFF, -255`
	Immediate supports parenthesis format for instructions with immediate offset.
	`addi x1, x0, 2` <=> `addi x1, 2(x0)`
	Immediate gets truncated to 20-bit or 12-bit based on instruction.
	6) Registers support different ABI names which are case-insensitive.
	7) `%hi()` and `%lo()` are assembly functions which can be used to extract most significant 20 bits and least significant 12 bits from a 32-bit symbol. This is useful to generate 32-bit address for load/store operations. Or load a 32-bit constant to a register. For eg:
	`# a4 ←load← from myvar in memory`
	`LUI a5, %hi(myvar)`
	`LW  a4, %lo(myvar)(a5)`
	
	`# a4 →store→ to myvar2 in memory`
	`LUI a5, %hi(myvar2)`
	`SW  a4, %lo(myvar2)(a5)`
	
	`# x1 ←load← “0xdeadbeef”`
	`LUI x1, %hi(0xdeadbeef)`
	`ADDI x1, x1, %lo(0xdeadbeef)`

	8) Supports labels for jump/branch instructions: ✓ Label is recommended to be of max. 16 ASCII characters ✓ Label should be stand-alone in new line for eg: `FIBONACC:` `mvi x1, 1` ✓ Label is case-sensitive. 9) Supports ASCII characters in the immediate values for instructions like MVI, LI. For eg: `MVI x0, ‘A’` # This is equivalent to MVI `x0, 0x41` Supports all 7-bit ASCII characters from 0x20 to 0x7E, ‘\n’, ‘\r’, ‘\t’.
Invoking Assembler	`pqr5asm.py -file=<assembly source file path> <-pcrel>` `-pcrel:` Applying this flag uses PC relative addressing for instructions like LA, JA. This helps in generating relocatable binary code. If this flag is not used, absolute address is loaded by the instructions. The binary code generated may not be relocatable.

1.1 Assembler flow

- Assembly code file → Validate all sections, linker directives → Initial formatting →
- Pre-processing: decode all labels and symbols and resolve all addresses → Resolve all assembly functions and immediates → Final formatting →
- Parse instructions line-by-line → Dump Binary code files on successful compilation.

2. Registers

Following registers are supported by the ISA and Assembler ABI.

Register Name	ABI Name	Description
`x0`	`x0`	Hard-wired Zero
`x1`	`ra`	Return Address
`x2`	`sp`	Stack Pointer
`x3`	`gp`	Global Pointer
`x4`	`tp`	Thread Pointer
`x5-x7`	`t0-t2`	Temporary Registers
`x8`	`s0/fp`	Saved Register/Frame Pointer
`x9`	`s1`	Saved Register
`x10-x11`	`a0-a1`	Function Arg/Return Val Registers
`x12-x17`	`a2-a7`	Function Arg Registers
`x18-x27`	`s2-s11`	Saved Registers
`x28-x31`	`t3-t6`	Temporary Registers

Table 2.1: Registers with ABI acronyms

3. Instructions

S No	Instruction	Syntax	Description
1.	**LUI**	`LUI rd, imm`	**Load Upper Immediate** Builds 32-bit constants. Loads 20-bit `imm[19:0]` into the upper 20-bit of `rd`. Loads the lower 12-bit of `rd` with zeroes. eg: `LUI x1, 0xFFF`
2.	**AUIPC**	`AUIPC rd, imm`	**Add Upper Immediate PC** Builds PC-relative addresses. Forms 32-bit offset from 20-bit `imm[19:0]` by loading into the upper 20-bit of `rd`, and loading the lower 12-bit with zeroes. Adds this offset to the PC, then places the result in `rd`.
Control Transfer Instructions			

3.	**JAL**	`JAL rd, label`
		OR
		`JAL rd, imm`
			Jump And Link
			Unconditional jump. Used to call subroutines. Stores the next instruction address, `pc+4` in `rd` for return from subroutine.
			20-bit `imm[19:0]` encodes signed offset in multiples of 2 bytes, and is added to the current pc to get the target address.
			target address =
			`pc + 32'(signed'({offset[20:1], 1'b0}))`
			The unconditional jump range = ±1 MB.

4.	**JALR**	`JALR rd, rs1, offset`	**Jump And Link Register** Unconditional Indirect jump. Used to call subroutines. Stores the next instruction address, `pc+4` in `rd` for return from subroutine. 12-bit `imm[11:0]` encodes signed offset, and is added to `rs1`, and clear 0th bit of result to get the target address. target address = `{(rs1 + 32'(signed'(offset)))[31:1], 1'b0}` The unconditional jump range = ±2 kB. (-2048 to +2047)

5.	**BEQ**	`BEQ rs1, rs2, label`
		OR
		`BEQ rs1, rs2, imm`
			Branch Equal
			Takes the branch if `rs1 == rs2`
			12-bit `imm[11:0]` encodes signed offset in multiples of 2 bytes, and is added to the current pc to get the target address.
			target address =

			`pc + 32'(signed'({offset[12:1], 1'b0}))` The conditional branch range = ±4 KB.

6.	**BNE**	`BNE rs1, rs2, label`
		OR
		`BNE rs1, rs2, imm`
			Branch Not Equal
			Takes the branch if `rs1 != rs2`

7.	**BLT**	`BLT rs1, rs2, label`
		OR
		`BLT rs1, rs2, imm`
			Branch Less Than
			Takes the branch if
			`signed'(rs1) < signed'(rs2)`

8.	**BGE**	`BGE rs1, rs2, label`
		OR
		`BGE rs1, rs2, imm`
			Branch Greater Than or Equal
			Takes the branch if
			`signed'(rs1) >= signed'(rs2)`

9.	**BLTU**	`BLTU rs1, rs2, label`
		OR
		`BLTU rs1, rs2, imm`
			Branch Less Than Unsigned
			Takes the branch if
			`rs1 < rs2`

10.	**BGEU**	`BGEU rs1, rs2, label`
		OR
		`BGEU rs1, rs2, imm`
			Branch Greater Than or Equal Unsigned
			Takes the branch if
			`rs1 >= rs2`

Load Store Instructions			
11.	**LB**	`LB rd, rs1, offset`	**Load Byte** Loads 8-bit data from memory, sign-extends to 32-bit, put into `rd`. load address = `32'rs1 + 32'(signed'(offset))` // expected to be 8-bit aligned
12.	**LH**	`LH rd, rs1, offset`	**Load Half-word** Loads 16-bit data from memory, sign-extends to 32-bit, put into `rd`.
13.	**LW**	`LW rd, rs1, offset`	**Load Word** Loads 32-bit data from memory, put into `rd`.
14.	**LBU**	`LBU rd, rs1, offset`	**Load Byte Unsigned** Loads 8-bit data from memory, zero-extends to 32-bit, put into `rd`.
15.	**LHU**	`LHU rd, rs1, offset`	**Load Half-word Unsigned** Loads 16-bit data from memory, sign-extends to 32-bit, put into `rd`.
16.	**SB**	`SB rs2, rs1, offset`	**Store Byte** Stores lower 8-bit of `rs2` in memory. store address =

			`32'rs1 + 32'(signed'(offset))` // expected to be 8-bit aligned
17.	**SH**	`SH rs2, rs1, offset`	**Store Half-word** Stores lower 16-bit of `rs2` in memory.
18.	**SW**	`SW rs2, rs1, offset`	**Store Word** Stores `rs2` in memory.
Integer Computation Instructions (ALU-I)			
19.	**ADDI**	`ADDI rd, rs1, imm`	**Add Immediate** `rd = rs1 + 32'(signed'(imm))` // overflow ignored

20.	**SLTI**	`SLTI rd, rs1, imm`	**Set Less Than Immediate**
			`rd = 1,`
			`if signed'(rs1) < 32'(signed'(imm)), else 0`

21.	**SLTIU**	`SLTIU rd, rs1, imm`	**Set Less Than Immediate Unsigned**
			`rd = 1,`
			`if rs1 < 32'(signed'(imm)), else 0`

22.	**XORI**	`XORI rd, rs1, imm`	**XOR Immediate** `rd = rs1 XOR 32'(signed'(imm))`
23.	**ORI**	`ORI rd, rs1, imm`	**OR Immediate** `rd = rs1 OR 32'(signed'(imm))`
24.	**ANDI**	`ANDI rd, rs1, imm`	**AND Immediate** `rd = rs1 AND 32'(signed'(imm))`
25.	**SLLI**	`SLLI rd, rs1, shamnt`	**Logical Left Shift Immediate** `rd = rs1 << shamnt[4:0]`
26.	**SRLI**	`SRLI rd, rs1, shamnt`	**Logical Right Shift Immediate** `rd = rs1 >> shamnt[4:0]`
27.	**SRAI**	`SRAI rd, rs1, shmant`	**Arithmetic Right Shift Immediate** `rd = signed' (rs1) >>> shamnt[4:0]`
Integer Computation Instructions (ALU-R)			
28.	**ADD**	`ADD rd, rs1, rs2`	**Add** `rd = rs1 + rs2` // overflow ignored
29.	**SUB**	`SUB rd, rs1, rs2`	**Subtract** `rd = rs1 - rs2` // underflow ignored
30.	**SLL**	`SLL rd, rs1, rs2`	**Logical Left Shift** `rd = rs1 << rs2[4:0]`
31.	**SLT**	`SLT rd, rs1, rs2`	**Set Less Than** `rd = 1,`

			if signed'(rs1) < signed'(rs2),
			else 0

32.	**SLTU**	SLTIU rd, rs1, rs2	**Set Less Than Unsigned**
			rd = 1,
			if rs1 < rs2, else 0

33.	**XOR**	XOR rd, rs1, rs2	**XOR** rd = rs1 XOR rs2
34.	**SRL**	SRL rd, rs1, rs2	**Logical Right Shift** rd = rs1 >> rs2[4:0]
35.	**SRA**	SRA rd, rs1, rs2	**Arithmetic Right Shift** rd = signed'(rs1) >>> rs2[4:0]
36.	**OR**	OR rd, rs1, rs2	**OR** rd = rs1 OR rs2
37.	**AND**	AND rd, rs1, rs2	**AND** rd = rs1 AND rs2
Pseudo/Custom Instructions			
38.	**MV**	MV rd, rs1 = ADDI rd, rs1, 0	**Move** rd = rs1
39.	**MVI**	MVI rd, imm = ADDI rd, x0, imm	**Move Immediate (12-bit immediate)** rd = imm
40.	**NOP**	NOP = ADDI x0, x0, 0	**No Operation**
41.	**J**	J label = JAL x0, label	**Plain Jump (short jump)** Jump to label
42.	**NOT**	NOT rd, rs1 = XORI rd, rs1, -1	**NOT** rd = NOT rs1
43.	**INV**	INV rd = XORI rd, rd, -1	**Invert** rd = NOT rd

44.	**SEQZ**	SEQZ rd, rs1
		= SLTIU rd, rs1, 1
			Set Equal to Zero
			rd = 1,
			if rs1 == 0, else 0

45.	**SNEZ**	SNEZ rd, rs2
		= SLTU rd, x0, rs2
			Set Not Equal to Zero
			rd = 1,
			if rs1 != 0, else 0

46.	**BEQZ**	BEQZ rs1, label = BEQ rs1, x0, label	**Branch Equal to Zero** Jump to label, if rs1 == 0, else 0
47.	**BNEZ**	BNEZ rs1, label = BNE rs1, x0, label	**Branch Not Equal to Zero**

			Jump to `label`, `if rs1 != 0, else 0`

48.	**LI**	`LI rd, imm **`
		`= LUI rd, U + ADDI rd, L`
			Load Immediate (32-bit immediate)
			`rd = imm`

49.	**LA**	`LA rd, label/symbol **`
		`= LUI rd, U + ADDI rd, L`
		With `-pcrel` flag:
		`= AUIPC rd, U + ADDI rd, L`
		If the address is encoded directly, for eg:
		`LA rd, 0xA0A0A0A0`
		Or if the reference is a data symbol, for eg:
		`LA rd, myvar`
		Absolute address is used always in this case even if `-pcrel` is set.
			Load Address
			`rd = address(label)`

50.	**JA**	`JA rd, label **`
		`= LUI rd, U + ADDI rd, L + JALR x0, rd, 0`
		With `-pcrel` flag:
		`= AUIPC rd, U + ADDI rd, L + JALR x0, rd, 0`
		If the address is encoded directly, for eg:
		`JA rd, 0xA0A0A0A0`
		Absolute address is used always in this case even if `-pcrel` is set.
			Load and Jump to Address (long jump)
			`rd = address(label)`

51.	**JR**	`JR rs1`
		`= JALR x0, rs1, 0`
			Jump Register Address
			Jump to address = `rs1`

Table 3.1: Instructions supported by Assembler

*** `U` and `L` are Upper 20-bit, `%hi(imm)` & Lower 12-bit, `%lo(imm)` values derived from 32-bit `imm`*

Conventions used:

`rd` = Destination register

`rs1` = Source register-1

`rs1` = Source register-2

`imm` = 12/20-bit immediate

4. Sections in the program

4.1 Text and data segments

Every assembly program is formatted as text and data sections. The text section, `.section .text`, encapsulates all the instructions. This forms the text segment of the program. The data section, `.section .data`, encapsulates all the symbols stored in the data memory. This forms the data segment of the program. Text section is mandatory in a program, while data section is not necessary. The data section should be defined before text section.

```
.section .data   # Data segment
<data symbols>

.section .text   # Text segment
<instructions>
```

4.2 Linker directives

Linker directives are used to map the different segments of a program to memory.

The directive `.org <addr>` is used to map the text and data sections. The `addr` is the base address of the segment to which the first instruction/data symbol is mapped. This directive is mandatory directive for any section and it should be 4-byte aligned. The directive should be defined immediately following the section. For eg:

```
.section .data      # Data segment
.org 0x40000000     # Data of the program is stored from this location
student:            # Data symbol student, addr = 0x40000000
.byte 1             # Data @(addr + 0) = 0x01, size = 1 byte
.word 95            # Data @(addr + 4) = 95, size = 4 bytes
.string “John Doe”  # String “John Doe” stored @(addr + 8), size = 9 bytes
.ascii ‘\n’         # Data @(addr + 17) = ‘\n’, size = 1 byte

.section .text   # Text segment
.org 0x00000000  # First instr of the program is stored in this location
<instructions>
```

If the assembler is configured to generate a relocatable program binary, the text segment can be loaded to a different base address than the one set by linker directive.

The directive `.p2align <alignment>` is used to force the alignment of a data symbol to 2^(alignment) bytes. This is useful to make the memory access efficient by making the data align with the native alignment of the processor. This directive should be defined immediately following the symbol declaration. For eg:

```
.section .data     # Data segment
.org 0x40000000    # Data of the program is stored from this location
city:              # Data symbol city, addr = 0x40000000
.string “London”   # String “London” stored @(addr + 0), size = 7 bytes
student:           # Data symbol student, addr = 0x40000007
```

```
.p2align 2          # Align student to 4 bytes by padding 1 zero byte
                    # The addr of student becomes 0x40000008
.byte 1             # Data @(addr + 0) = 0x01, size = 1 byte
.word 95            # Data @(addr + 4) = 95, size = 4 bytes
.string "John Doe"  # String "John Doe" stored @(addr + 8), size = 9 bytes
.ascii '\n'         # Data @(addr + 17) = '\n', size = 1 byte
```

4.3 Labels

Labels are used in the program to mark specific points in the code for reference purposes, making the code easier to maintain. They serve as a way to identify locations within a program, often for branching, looping, or jumping purposes. For eg:

```
.section .text   # Text segment
.org 0x00000000  # First instr of the program is stored in this location

START:  # Start of a program
<instruction 1>
```

4.4 Symbols

Variables used in the program are stored in the data memory. They are referred by the program instructions using symbols. The symbol represents the base address of the variable. A symbol can have a set of contiguous data under it of different data types.

Data type	Usage	Description
.byte	.byte 0xFF .byte 255	Byte, size = 1 byte Supports initializing with unsigned/signed integer, hex value
.hword	.hword 0xABCD	Naturally-aligned half-word, size = 2 bytes Supports initializing with unsigned/signed integer, hex value
.word	.word 0xABCDEF01	Naturally-aligned word, size = 4 bytes Supports initializing with unsigned/signed integer, hex value
.ascii	.ascii 'A'	Char byte, size = 1 byte Supports initializing with a single ASCII character enclosed within single quotes. Supports all 7-bit ASCII characters from 0x20 to 0x7E, '\n', '\r', '\t'.
.string	.string "Hello"	Null-terminated string, size = <string size> + 1 byte Supports initializing with ASCII characters enclosed within double quotes. Supports all 7-bit ASCII characters from 0x20 to 0x7E, '\n', '\r', '\t'.
.zero	.zero 4	Appends specified no. of zero bytes.

4.5 BSS segment

BSS segment is not directly supported. The user can however emulate a BSS segment in a program by defining uninitialized symbols under data section and explicitly initializing them to zero using `.zero` keyword. For eg:

```
myvar:
.zero 4 # myvar is a symbol in memory of size 4 bytes & initialized to 0
```

4.6 Stack and other segments

The user should explicitly load the stack pointers if required, and other memory pointers with a start-up code. Currently, there are no assembly/linker directives to support these segments.

5. Binary/Hex file format

Input to the assembler is the assembly code file in **.s** file format. On successful compilation and linking, following files are generated by the assembler:

1. Binary code files for instructions and data in **.bin** format, which may be decoded and loaded to instruction and data memories of CPU to boot and execute.
2. Binary/Hex code text files for instructions and data in ASCII **.txt** format. This is human readable.

The **.bin** file contains the instructions to be executed/data symbols as sequence of bytes in binary format.

The following example shows how an instruction binary file, *sample_imem.bin* file, and a data binary file, *sample_dmem.bin* are encoded. Big Endian format is used in the binary file.

```
// sample_imem.bin
<0xF0><0xF0><0xF0><0xF0>   # Pre-amble marks the start
<0x00><0x00><0x00><0x28>   # Program size = (no. of instr, N x 4) bytes
<baB3><baB2><baB1><baB0>   # Base address of the program byte[3] to [0]
<inB3><inB2><inB1><inB0>   # Instruction-1 byte[3] to [0]
<inB3><inB2><inB1><inB0>   # Instruction-2 byte[3] to [0]
….
….
<inB3><inB2><inB1><inB0>    # Instruction-N byte[3] to [0]
<0xE0><0xE0><0xE0><0xE0>    # Post-amble marks the end

// sample_dmem.bin
<0xF0><0xF0><0xF0><0xF0>   # Pre-amble marks the start
<0x00><0x00><0x00><0x28>   # Data size = (no. of words, N x 4) bytes
<baB3><baB2><baB1><baB0>   # Base address of the data section byte[3] to [0]
<wdB3><wdB2><wdB1><wdB0>   # Word-1 byte[3] to [0]
<wdB3><wdB2><wdB1><wdB0>   # Word-2 byte[3] to [0]
….
….
<wdB3><wdB2><wdB1><wdB0>    # Word-N byte[3] to [0]
<0xE0><0xE0><0xE0><0xE0>    # Post-amble marks the end
```

PQR5 Assembler

An open-source RISC-V Assembler for RV32I ISA

Developer	: **Mitu Raj**
Vendor	: **Chip**munk **Logic**™, *chip@chipmunklogic.com*
Website	: **chipmunklogic.com**