RISC-721-Assembler User's Guide

Connor Goldberg

Introduction

This assembler is specifically designed for my custom RISC-721 CPU design. The specifications for my design are as follows:

- Register/register data transfer
- 32-bit data words
- 16-register deep hardware stack
- Memory-mapped I/O
- More: view the instruction set architecture

Installation

To install, download all of the python files into the same directory. This assumes that Python 2.7 is already installed. This does not depend on any 3rd party Python modules.

```
$ git clone https://github.com/connorjan/RISC-721-Assembler.git
```

Execution

Open either a command prompt (windows) or a terminal (Mac / Linux). Change directory into the directory where the python files were downloaded.

Basic usage: `$ python Assembler.py [-h] [-o out-file] assembly-file.asm`

By default, the output file takes the name of the input file, but replacing the original file extension with `'.mif'`. If there are any specified constants that are placed into data memory then a separate mif file will be created for the data memory that takes the form: `output-file-name_DM.mif` by default. For instance if an input file named `Foo.asm` was specified without specifying an output file, the assembler would assemble the file into `Foo.mif` and `Foo_DM.mif` (if applicable). The name of the output file can be overridden by using the `-o` or `--output` option in the command line. Example:

```
$ python Assembler.py cjg.asm --output cjg_new.mif
```

would result in an output program memory file of `cjg_new.mif` and a data memory file (if applicable) of `cjg_new_DM.mif`.

Assembly Language

The assembly language currently has support for 4 different sections: includes, constants, directives, and code. Instructions and assembler keywords/directives are delimited by line therefore no semicolons are required (but they won't hurt). There are no block comments, however a line comment can be started by using C-style line comments: `//`

Mathematical Expressions

There are several places in the assembly code where mathematical expressions can be used. This section outlines the restrictions on the formatting for these expressions.

- Standard order of operation rules apply
- The expressions are evaluated by the Python interpreter, therefore expressions take on Python syntax
- The operations possible are:
  - Addition, Subtraction, Multiplication, Division, Exponent, Modulus, Logical Shift, Logical AND, Logical OR, Logical NOT
- Values can be in decimal or hexadecimal – hexadecimal values must start with `0x`

Constants Section

This section is started by the identifier of `.constants` and ends with the line: `.endconstants`. This section contains values that will be placed into memory. The initializations take the form of:

```
destination-address = value
```

Both the address and value fields can contain mathematical expressions (explained in detail in the section above). Additionally, the value field can contain an additional directive for floating point numbers. These numbers are denoted by the `.float(EXPRESSION)` directive. The value will be replaced with the 32-bit, single precision IEEE formatted value.

Example:

```
1   .constants
2       0x10 = 20
3       32 = 0xBEEF
4       0x30 = .float(1)
5       0x40 = .float(-2.0/4)
6   .endconstants
```

Directives Section

This section is started by the identifier of `.directives` and ends with the line `.enddirectives`. Within this section it is possible to create pre-processor-like definitions for variables. These variables are never seen in the actual machine code, they are replaced by the assembler by their actual values before the code is assembled. Additionally, the directives are evaluated line-by-line therefore you can use previously defined directives to define a new one on a later line.

Example:

```
1   .directives
2       foo = 0x01
3       bar = 0x02
4       three = foo + bar
5       nine = three^2
6       statusRegister = R0
7   .enddirectives
```

Includes Section

This section is started by the identifier of `.includes` and ends with the line `.endincludes`. Within this section, multiple assembly files can be specified that will all be assembled into a single program. The format for adding include files includes a label and a path to the file to include. The label is available to the code section as it contains the first address of code in that specific include file. The filename parameter cannot contain spaces.

When including a file, constants from across all files will be placed into the single data memory file, however labels and directives are only available to each individual file. This allows labels of the same name to be used in different files without conflict.

Example:

```
1   .includes
2       multiply = multiplication.asm
3       divide = division.asm
4   .endincludes
```

Code Section

This is denoted by the `.code` and `.endcode` keywords. In this section, the instructions are decoded to machine code. Each original line of assembly code will be preserved and put in the memory file as a comment for reference and as a debugging aid.

The instructions take either one, two, or three operands depending on the instruction. The general syntax for these, respectively, is as follows:

```
1   .code
2       mnemonic destination/source
3       mnemonic destination, source
4       mnemonic destination, source, source
5   .endcode
```

There are 6 main types of instructions which each have their own syntax:

- Load/Store
- Data Transfer
- Flow Control
- Manipulation
- Rotate/Shift
- Floating Point

Some instruction mnemonics that end in a 'C' represent an instruction where one of the operands is a constant. For example: `CPYC` is the mnemonic for copying a constant value into a register. The constant values are limited to 16-bits (`65535` or `0xFFFF`).

Load/Store Instructions

Load and store instructions are responsible for loading a register with a value from memory or a memory-mapped input peripheral as well as storing a value into memory or outputting to an output peripheral. The memory-mapped peripherals can be found in the instruction set architecture.

There are four addressing modes available:

Addressing Mode	Description
Register Direct	The address is the value of a register
Absolute	The address is a constant
Indexed	The address is a register + a constant
PC Relative	The address is the PC + a constant

The instructions available are as follows:

Instruction	Mnemonic	Function
Load	LD	Loads a value into a register from memory or an input
Store	ST	Stores a value from a register into memory or an output

Example 1:

```
1   .code
2       LD R1, M[R1] // Register Direct
3       LD R2, M[0x100] // Absolute
4       LD R3, M[R1 + 0x100] // Indexed
5       LD R4, M[0x100 + PC] // PC Relative
6   .endcode
```

In *Example 1* some registers are loaded from memory using different addressing modes.

Example 2:

```
1   .directives
2       LED = 0x3FFF
3   .enddirectives
4
5   .code
6       LD R1, 0x10
7       ST M[LED], R1
8   .endcode
```

In *Example 2* a value is loaded into R1 from address 0x10, then the value is written to the LEDs using the `store` instruction where the LEDs are memory mapped to address `0x3FFF`.

Data Transfer Instructions

Data transfer instructions are responsible for moving data between registers, or between registers and the stack. For any of the instructions, a 'C' character can be appended to the instruction which will allow a constant to be used as the last source operand instead of a register value. The constant has a max size of 16-bits.

Instruction	Mnemonic	Function
Copy	CPY	Copies a register's value from the source to the destination
Push	PUSH	Pushes the source onto the top of the stack
Pop	POP	Pops the top of the stack into the destination

Example:

```
1   .code
2       CPYC R1, 0x100
3       CPYC R2, 0x200
4       PUSH R1
5       PUSH R2
6       POP R3
7       POP R4
8       CPY R5, R4
9   .endcode
```

In this example, first the constant value of 0x100 is copied to R1, then 0x200 is copied into R2. Then R1 (0x100) is pushed onto the top of the stack followed by R2 (0x200). Then the top of the stack (0x200) is popped into R3 and then the new top of stack (0x100) is popped into R4 leaving the stack empty. Finally, the value in R4 (0x100) is copied into R5.

Manipulation Instructions

Manipulation instructions are responsible for manipulating data within registers and make use of the arithmetic logic unit (ALU). For any of the instructions, a 'C' character can be appended to the instruction which will allow a constant to be used as the last source operand instead of a register value. The constant has a max size of 16-bits.

Instruction	Mnemonic	Function	Status Register Bits Set
Addition	ADD	Destination $\leftarrow Source_1 + Source_2$	C,N,V,Z
Subtraction	SUB	Destination $\leftarrow Source_1 + Source_2$	C,N,V,Z
Logical AND	AND	Destination $\leftarrow Source_1$ & $Source_2$	C,N,V,Z
Logical OR / Bit Set	OR / BIS	Destination $\leftarrow Source_1 \mid Source_2$	C,N,V,Z
Bit Clear	BIC	Destination $\leftarrow Source_1$ & $\sim Source_2$	C,N,V,Z
Logical Exclusive OR	XOR	Destination $\leftarrow Source_1 \hat{} Source_2$	C,N,V,Z
Logical Invert	NOT	Destination $\leftarrow \sim Source_1$	C,N,V,Z
Logical Compare	CMP	$Source_1$ - $Source_2$	C,N,V,Z,GE,L

Example:

```
1   .code
2        ADD  R1, R2, R3 // Add R2 and R3 and store the result in R1
3        SUBC R1, R1, 0x1 // Subtract 1 from R1, then store the result in R1
4        ANDC R2, R2, 0xFF // AND R2 and 0xFF then store the result in R2
5        BISC R3, R3, 0x1234 // OR R3 with 0x1234
6        BICC R4, R4, 0x00FF // AND R4 with ~0x00FF and store the result in R4
7        XOR R5, R3, R4 // XOR R3 with R4 and store the result in R5
8        NOT R6, R5 // Invert R5 and store the result in R6
9        CMP R5, R6 // Subtract R6 from R5 but dont set a result, only set SR bits
10
11  .endcode
```

In the example above, the comments describe the operation of each line.

Rotate/Shift Instructions

Rotate/shift instructions are a subset of the ALU instructions dedicated to rotating and shifting data within a register. Similar to the manipulation instructions, a 'C' character can be appended to the instruction which will allow a constant to be used as the last source operand instead of a register value, however the max constant for these instructions is only 6-bits.

Mnemonic	Function
SRL	Shift Right Logical
SLL	Shift Left Logical
SRA	Shift Right Arithmetic
RTR	Rotate Right
RTL	Rotate Left
RRC	Rotate Right Through Carry
RLC	Rotate Left Through Carry

Example:

```
1   .code
2       CPYC R1, 0x1 // Set R1 to 0x1
3       RRC R1, R1, R1
4       RLCC R1, R1, 0x2
5   .endcode
```

In the example above, line 3 rotates R1 right through carry by R1 (0x1) and stores the result in R1. After this instruction executes the value of R1 would be 0x0 and the carry bit in the status register would be set. Then line 4 rotates R1 left through carry by 0x2 and stores the result in R1. This would result in R1 being set to 0x2, and the carry bit unset.

Flow Control Instructions

Flow control instructions are responsible for changing the program execution flow, usually based upon the result of a previous instruction and the current state of the processor. Most flow control instructions only require a single operand: a label. A label must be placed on the same line as an instruction before the mnemonic, and the label must end in a colon.

Example:

```
1   .code
2       init:   CPYC R1, 0x0
3               CPYC R2, 0x0
4   .endcode
```

In this example, 'init' is the label. Labels are always case sensitive and cannot contain any whitespace.

The conditional jump instructions operate based on the bits that are set in the status register (information on the status register is located in the Appendix).

Instruction	Mnemonic	Function
Jump Unconditional	JU or JMP	Jumps to label unconditionally
Jump if carry	JC	Jumps to label if the carry bit is set
Jump if not carry	JNC	Jumps to label if the carry bit is not set
Jump if negative	JN	Jumps to label if the negative bit is set
Jump if not negative	JNN	Jumps to label if the negative bit is not set
Jump if overflow	JV	Jumps to label if the overflow bit is set
Jump if not overflow	JNV	Jumps to label if the overflow bit is not set
Jump if zero	JEQ or JZ	Jumps to label if the zero bit is set
Jump if not zero	JNE or JNZ	Jumps to label if the zero bit is not set
Jump if greater or equal	JGE	Jumps to label if the greater bit is set
Jump if less than	JL	Jumps to label if the less bit is set
Call	CALL	Calls the function starting at the label
Return	RET	Returns from the function to the address proceeding the call
Return from interrupt	RETI	Returns from an interrupt service routine

Example:

```
1   .code
2       init:   CPYC R1, 0x1 // Set R1 to 0x1
3               CPYC R2, 0x1 // Set R2 to 0x1
4               CALL compare // Call the compare function
5               JU init // Jump to init
6
7       compare: CMP R1, R2 // Subtract R2 from R1
8                JEQ equal // Jump if equal
9
10      diff:   CPYC R3, 0x1 // Set R3 to 0x1
11              RET // Return from call
12
13      equal:  CPYC R3, 0x0 // Set R3 to 0x0
14              RET // Return from call
15  .endcode
```

The example above compares R1 and R2. If they are different, then R3 is set to 0x1, otherwise R3 is set to 0x0. This is a very trivial function however it demonstrates labels, calls, returns, and jump instructions.

Floating Point Instructions

Floating point instructions make use of the floating point unit (FPU) to perform 32-bit, single precision IEEE floating point instructions on two register operands.

Mnemonic	Function
FA	Floating Point Addition
FS	Floating Point Subtraction
FM	Floating Point Multiplication
FD	Floating Point Division

Floating point instructions cannot be used with in-line constant values, however a constant memory value can be initialized with the `.float` directive which then can be loaded into memory.

Example:

```
1   .constants
2       0x1 = .float(10)
3       0x2 = .float(5)
4   .endconstants
5   .code
6       LD R1, M[0x1] // Load R1 with .float(10)
7       LD R2, M[0x2] // Load R2 with .float(5)
8       FA R3, R1, R2
9       FM R4, R1, R2
10  .endcode
```

The example above first loads the floating point constants from memory into registers. The values are then added and the result is stored into R3. The values are then multiplied and the result is stored into R4.

Emulated Instructions

There are several emulated instructions built into the assembler. These instructions are not actual instructions known by the processor, but shortcuts to commonly used instructions with common operands.

Mnemonic	Emulated Instruction	Function
INC R_i	ADDC R_i, R_i, 0x1	Increment R_i by 1
DEC R_i	SUBC R_i, R_i, 0x1	Decrement R_i by 1
CLR R_i	SUB R_i, R_i, R_i	Clear R_i to 0
CLRC	BICC R0, R0, 0x1	Clear the carry bit of the status register
NOP	CPY R1, R1	No operation

Interrupts

General Specifications

There are several interrupts that can be used where each interrupt requires its own interrupt service routine (ISR). ISR0-3 are reserved for 4 GPIO pins, ISR4 is an internal timer, and ISR5 is for the instruction counter. Each ISR must manually be enabled by setting the enable bits within the status register (explained in the Appendix). More information can also be found in the instruction set architecture.

The interrupts are active-high and only activate on the rising edge of the signal, and will only activate once until the signal returns to a low state. This prevents excessive interrupts when using the GPIO pins as a physical input to the system where the signal may be high for an extended period of time.

When within an interrupt service routine, registers 0 - 23 are copied to a shadow register file, then are restored once the interrupt is serviced. This is to guarantee data consistency even in the case of an interrupt. Registers 24-31 are not saved however, and can be used to get data out of an ISR (*e.g.* setting a software flag).

Usage

To use the interrupts they first must be enabled which can be done using a bit set (`BIS`) operation. The ISRs are denoted by a special label format: `ISR_x` where x is the ISR number from 0-7. The ISR *must* end with the return from interrupt instruction (`RETI`) otherwise there will be undefined behavior. The assembler will find the interrupt labels and place their addresses into the interrupt vector table which is located at the end of the program memory.

Example:

```
1   .directives
2       SWFlag = R24
3       EnableISR0 = 1<<6
4   .enddirectives
5   .code
6       init:       CLR SWFlag // Clear the SWFlag (R24)
7                   BISC R0, R0, EnableISR0 // Enable ISR0
8
9       loop:       CMPC SWFlag, 0x1 // Check if the SWFlag is set
10                  JNE loop // If it's not, try again
11
12      pressed:    CPYC R1, 0x1 // Else, set R1 to 1
13                  JU init // Jump back to init
14
15      ISR_0:      CPYC SWFlag, 1 // Set the SWFlag to 1
16                  RETI // Return from interrupt
17  .endcode
```

The above example is a trivial case of how to set up an interrupt. The location of the `SWFlag` variable is located at R24 (line 2) because that is one of the registers that is not saved in the shadow register file. Alternatively, the flag could have been stored in memory.

Appendix

Status Register

The status register (SR) is the register located at R0. This register is reserved for special purposes including storing the status of various ALU instructions, as well as providing conditions for the flow control instructions and the enable bits for interrupts. The register holds the carry, negative, overflow, zero, greater than or equal*, and less than* bits. These status bits are set by the ALU depending on the result of the operation.

> *These bits are only set when using the compare (CMP) instruction

The SR also is responsible for the interrupt service routine (ISR) enable bis which must be enabled for that specific ISR to be activated. The ISRs are controlled by bits [13:6] in the SR.

More information about the status register can be found in the instruction set architecture.