



main ▾

StackCPU / README.md



dervish77 Update README.md



4e6ecb4 · 1 hour ago



528 lines (376 loc) · 23.5 KB

Preview

Code

Blame

Raw



Stack CPU

Custom stack-based CPU design

StackCPU is a custom CPU design that utilizes a stack-based memory model instead of the usual memory models seen in processors such as 6502 or Z80. The StackCPU project was undertaken mostly as an exercise in virtual CPU design that could be implemented via a simulator and utilize custom assembler tools. The StackCPU design definition has a limited memory size (4K bytes) to simplify any hardware implementations using the specification. The StackCPU design should be simple enough to be implemented in hardware via an FPGA or perhaps even using discrete TTL devices.

The stack-based design for StackCPU uses the stack for nearly all operations. Thus data is constantly moved on and off the stack during normal operation. Some instructions use a combination of the stack and the accumulator register (i.e. math and logic operations). The data memory region is used to store and retrieve temporary variables when stack operations might result in those values being lost or destroyed. While the accumulator register (AC) is used for arithmetic and logic operations, it is not directly accessible via any machine instructions. Likewise the data register (DR) is used to perform transfers to and from the data memory region, the register itself is not directly accessible via any machine instructions. Finally, the temp register (TR) is also used for math and logic operations, and is not directly accessible.

The StackCPU design includes registers for getting input from the outside world (IR) and (SR) and sending output to outside world (OR) and (PR). The IR and OR registers are used to input and output of numerical values directly, or use them to read/write control signals for external devices. The SR and PR registers are used to input and output ascii characters from/to a serial terminal via an RS232 port.

The StackCPU design's memory model is limited to 4K bytes, thus the external address bus is 12 bits. Additional memory could be accessed by using 1 to 4 OR register pins as additional address pins. The memory model is further divided into three sections - program, data, and stack. By default, the program section starts at address 0x0000 and is 3K bytes in size. By default, the data section starts at address 0x0C00 and is 768 bytes in size. The stack section starts at address 0x0F00 and is 256 bytes in size. Note that the program and data sections are flexible with respect to their division if more or less program space is required (of course by also adjusting the space allocated to the data space). The stack section, however, is fixed and cannot be expanded or contracted.

All of the StackCPU design's internal registers are 16 bits. But the external data bus is only 8 bits, so when reading data from memory into any internal register, that data will typically only occupy the lower half of the internal register (unless a specific instruction takes care of loading both halves of the internal register). The memory interface also includes a few control lines such as memory clock, output enable and write select.

Repo contents:

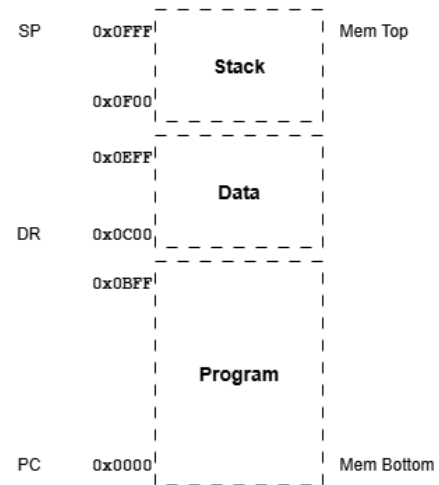
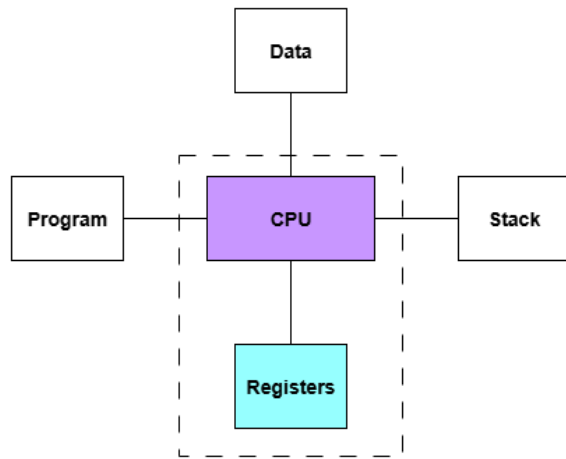
- docs/ - system design docs, block diagrams, etc
- examples/ - example code using the Stack CPU
- hw/ - hardware design files
- ref/ - reference documents for integrated packages
- src/ - source code for the Stack CPU tools

Requirements

- StackCPU shall support TBD.

High Level Design

Machine Model



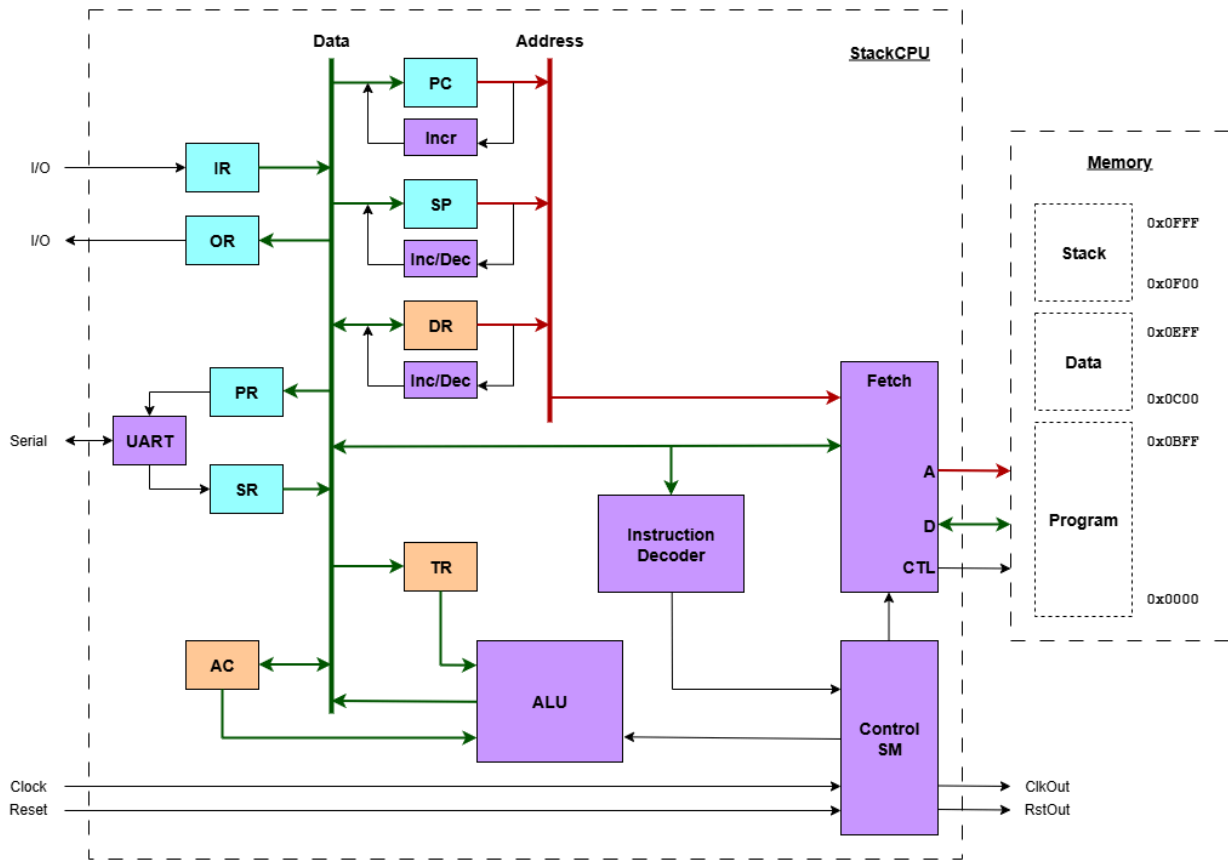
Register Model

While all of the StackCPU registers are internally implemented as 16 bit registers, several of them only utilize the lower byte and the upper byte is ignored. Namely, the program counter (PC), the stack pointer (SP), and the data register (DR) are all full 16 bit registers. All the remaining registers only utilize the lower byte. Note that several registers are not directly accessible by the programmer, i.e. they are considered "hidden" registers.

PC	program counter - grows up from <bottom of mem>
SP	stack pointer - grows down from <top of mem>
DR	data register (hidden) - points to <mem above top of program>
AC	accumulator (hidden) - used for math/logic operations
TR	temp register (hidden) - used for math/logic operations
IR	input register - external input
OR	output register - external output
SR	serial register - external input (serial data input)
PR	print register - external output (serial data output)



Architecture Diagram



Instruction Model

StackCPU instruction sizes range from one byte to three bytes. Two byte instructions have a single operand which is a direct data value to be loaded into a register or loaded into the stack. Three byte instructions have two operands which in all cases these operands form a memory address that is loaded into either the data register (DR) or into the program counter (PC).

```

OPERATION
OPERATION <direct operand>          #dd or "c"
OPERATION <memory address>          $hhhh or %variable
OPERATION <label>                    &label

```



Instruction Set Architecture

Transfer Instructions

```

---          instruction fetch          fetch op_code ->
<inst dec>

PSH <do>     push direct data to top of stack          fetch op -> AC

```



		push AC
PSA	push AC to top of stack	push AC
POP	pops top of stack	pop -> AC
LDM <mem>	loads data from memory to top of stack	fetch op1 -> DRH fetch op2 -> DRL M[DR] -> AC push AC
LDI	increments DR, load data mem to TOS	DR + 1 -> DR M[DR] -> AC push AC
LDD	decrements DR, load data mem to TOS	DR - 1 -> DR M[DR] -> AC push AC
STM <mem>	stores data from top of stack to memory	fetch op1 -> DRH fetch op2 -> DRL pop -> AC AC -> M[DR]
STI	increments DR, stores TOS to data mem	DR + 1 -> DR pop -> AC AC -> M[DR]
STD	decrements DR, stores TOS to data mem	DR - 1 -> DR pop -> AC AC -> M[DR]

Math Instructions

ADD	adds top two stack values (add replaces top 2 stack with sum)	pop -> TR pop -> AC AC = AC + TR push AC
SUB	subtracts top two stack values (sub replaces top 2 stack with diff)	pop -> TR pop -> AC AC = AC - TR push AC
NEG	negates top of stack	pop -> AC 0 -> TR AC = TR - AC push AC



LSR logical shift top of stack right

pop -> AC
AC = AC >> 1
push AC

LSL logical shift top of stack left

pop -> AC
AC = AC << 1
push AC

Logical Instructions

AND <do> AND top of stack with data

fetch op -> TR
pop -> AC
AC = AC & TR
push AC

ORR <do> OR top of stack with data

fetch op -> TR
pop -> AC
AC = AC | TR
push AC

XOR <do> XOR top of stack with data

fetch op -> TR
pop -> AC
AC = AC ^ TR
push AC

INV Invert top of stack

pop -> TR
AC = invert TR
push AC

Compare/Branch Instructions

CPE <do> compare if top of stack is equal

fetch op -> TR
pop -> AC
push AC
if AC equal TR,
 push 0
else,
 push 1


CNE <do> compare if top of stack is not equal

fetch op -> TR
pop -> AC
push AC
if AC not equal


TR,

		push 0 else, push 1
BRZ <label>	branch if top of stack is zero	fetch op1 -> DRH fetch op2 -> DRL pop -> AC if AC equal to 0, DR -> PC
BRN <label>	branch if top of stack is not zero	fetch op1 -> DRH fetch op2 -> DRL pop -> AC if AC not equal to 0, DR -> PC
BRU <label>	branch unconditionally	fetch op1 -> DRH fetch op2 -> DRL DR -> PC

I/O instructions

INP	inputs I/O to top of stack	IR -> AC push AC	
OUT	outputs top stack to I/O	pop -> AC AC -> OR	
SER	inputs serial to top of stack	SR -> AC push AC	
PRT	outputs top of stack to serial	pop -> AC AC -> PR	

Special instructions

NOP	no operation	no state change	
CLS	clear the stack	<mem top> -> SP	
END	end of program (aka HALT)	PC - 1 -> PC	
RST	reset cpu	0 -> AC 0 -> OR 0 -> PR	

DR

<data start> ->

<mem top> -> SP

<mem bot> -> PC

Notes

"fetch"

M[PC] -> <dest>

PC + 1 -> PC



"push"

SP - 1 -> SP

<source> -> M[SP]

"pop"

M[SP] -> <dest>

SP + 1 -> SP

[Instruction op-code details](#)

Examples

Hello

```
; prints Hello
.ORG 0x0000
.DAT 0x0C00
:start CLS
      PSH "H"
      PRT
      PSH "E"
      PRT
      PSH "L"
      PRT
      PSH "L"
      PRT
      PSH "O"
      PRT
      PSH #13      ; CR
      PRT
      PSH #10      ; LF
      PRT
      END
```



Add Numbers from 1 to 5


```

; Adds numbers from 1 to 5, outputs sum
.ORG 0x0000
.DAT 0x0C00
:start  CLS
        PSH #1
        PSH #2
        ADD
        PSH #3
        ADD
        PSH #4
        ADD
        PSH #5
        ADD
        OUT
        END

```



Add Numbers from 1 to 10 in a Loop

```

; Adds numbers from 1 to 10 in a loop, outputs sum
.ORG 0x0000
.DAT 0x0C00
.EQU 0x0C00 %sum
:start  CLS
        PSH #0
        STM $0C00      ; clear sum in memory (use direct address ref)
        NOP
        PSH #0          ; push "last" value on stack
        NOP
; create list of numbers on stack
        PSH #1
:iloop  POP              ; pop the "next" number
        PSA              ; put it back on stack
        PSA              ; push a copy
        PSH #1           ; push incr value
        ADD              ; add replaces top 2 stack values with sum
        CPE #10
        BRN &iloop       ; loop to push next incr value on stack
        NOP
; sum the numbers on the stack
:aloop  LDM %sum          ; use variable ref
        ADD
        STM %sum          ; store sum to memory
        CPE #0
        BRN &aloop       ; loop to add next num to sum
        NOP
; output results
        LDM $0C00

```



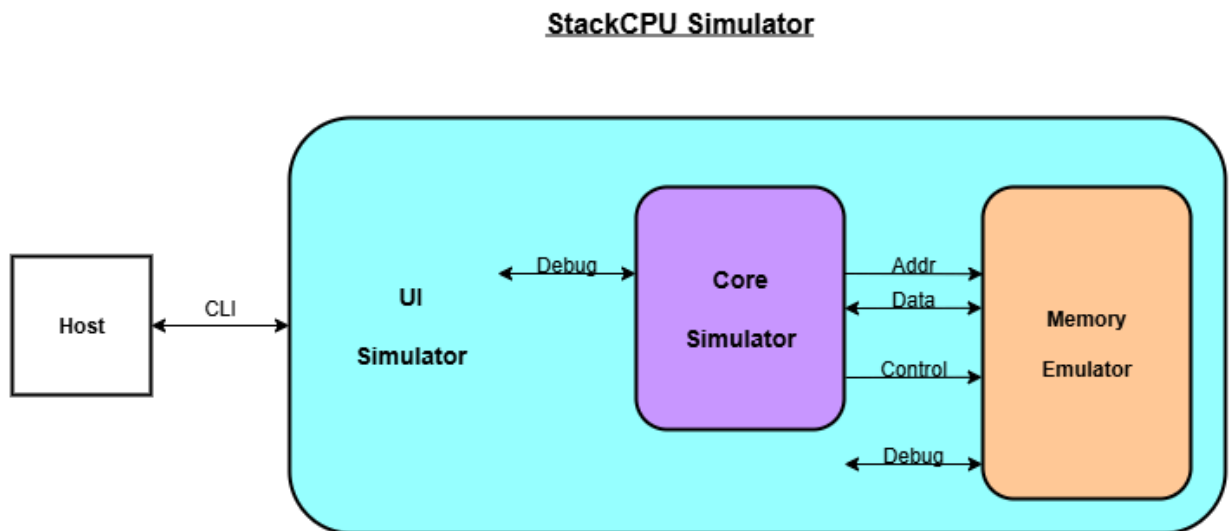
```
OUT          ; output sum
END
```

Detailed Design

Software

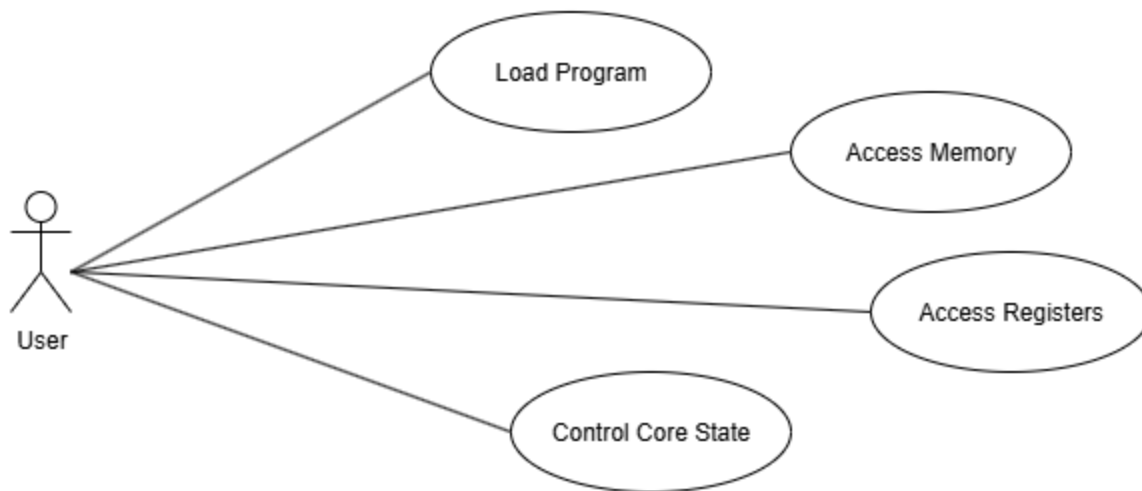
Simulator

The StackCPU Simulator is a simulation of the StackCPU implemented in C/C++ and Python that can be executed on Windows (via Cygwin) or Linux environments. The simulator is divided into three parts - UI Simulator, Core Simulator, and Memory Simulator. The UI Simulator wraps around the Core Simulator to provide a view into and control over the internals of the StackCPU device. The Core Simulator runs the core machine model that executes StackCPU machine code. The Memory Simulator provides an emulation of the system memory that is accessed by the StackCPU device during runtime.



The StackCPU Simulator allows the user to load programs into memory, access memory locations, access CPU registers, and control system states. This access and control is provided by the UI Simulator portion of the StackCPU Simulator.

StackSim Use Cases



UI Simulator

The UI simulator is a CLI interface that enables the loading of binary images into the Memory simulator, access and control over registers in the Core simulator, and control over execution of code stored in memory. The host interface of the UI simulator is provided in two parts -- command line arguments and the CLI itself.

Command line arguments:

Usage: `stacksim [options]`



options:

- `-d filename` - output debug data to log "filename" (default is off)
- `-f filename` - load memory with data from "filename" (default is "file.bin")
- `-m <mode>` - enter <mode> on startup
 where <mode> is 0 for idle, 1 for halt (default), 2 for run, 3 for single step
- `-h` - display command arguments
- `-v` - display version

Example: `stacksim -m 1 -f myprog.bin`

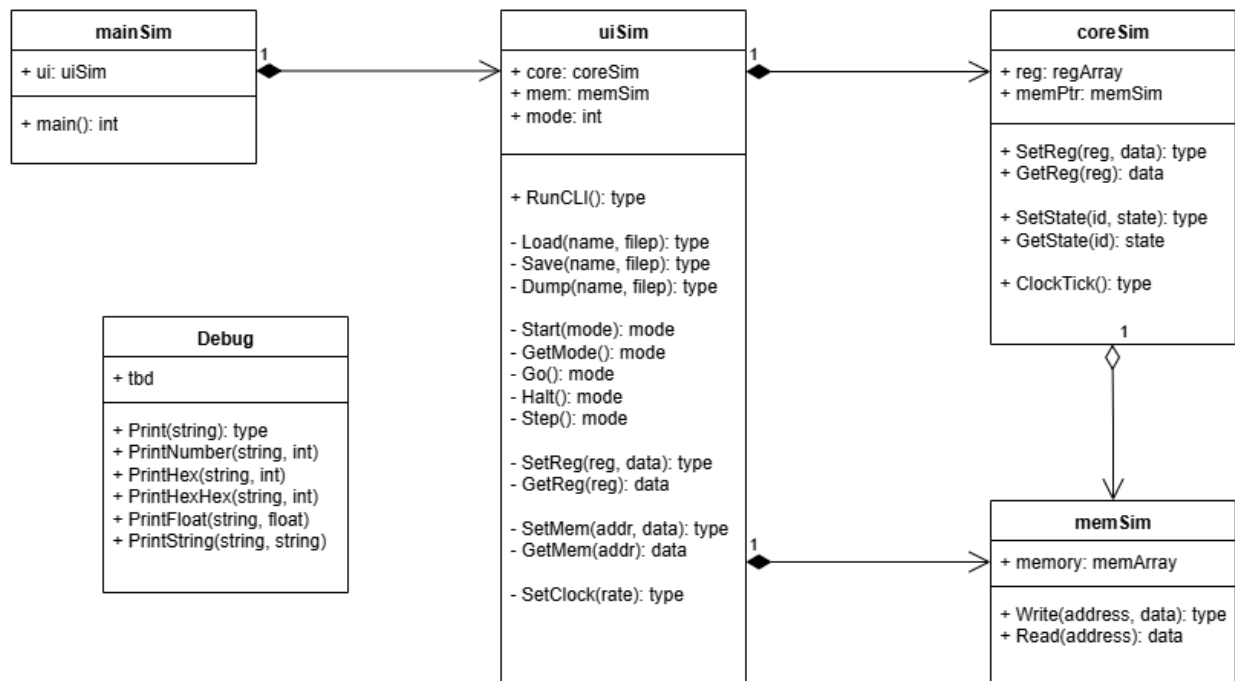
The CLI commands:

- `l filename` - load binary "file" into memory simulator
- `s filename` - save memory simulator to binary "file"



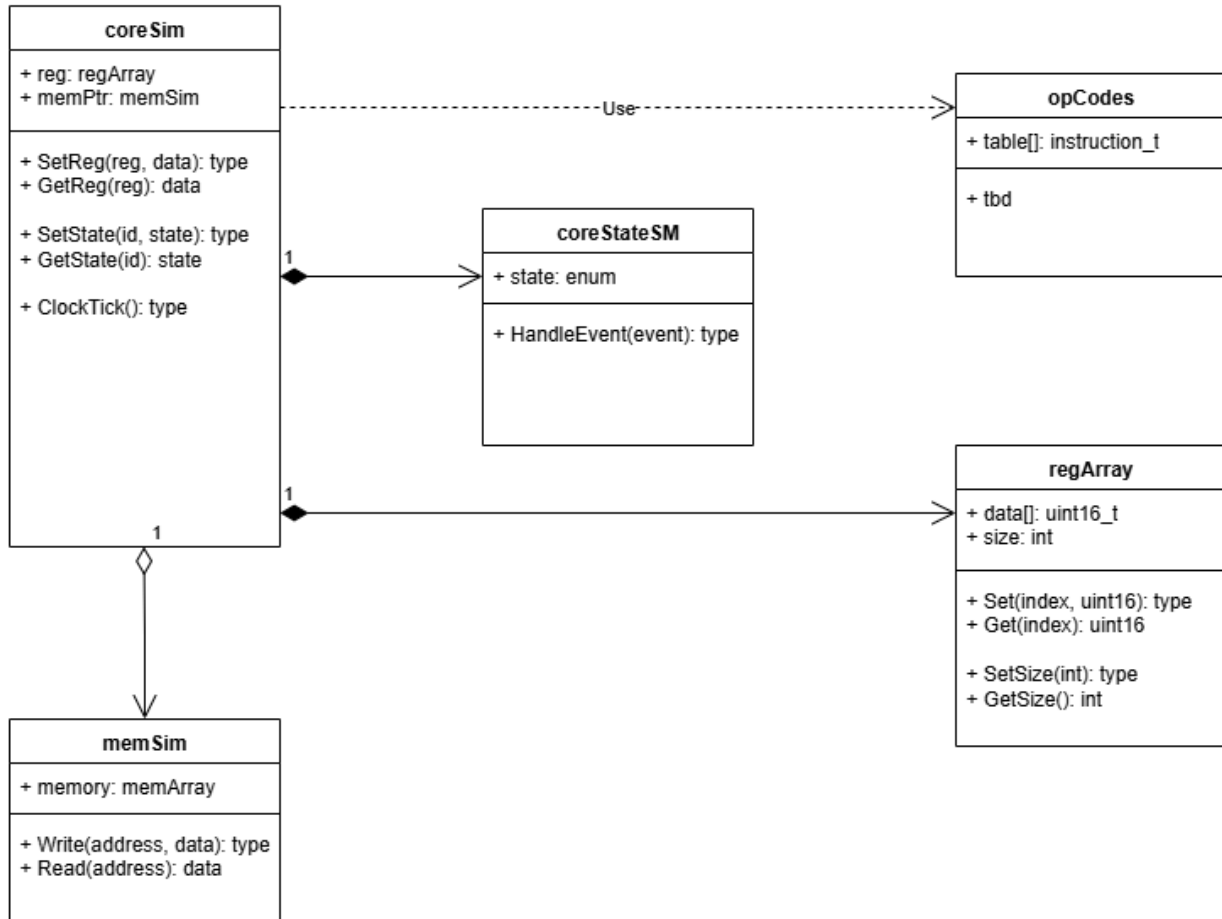
d filename	- dump memory to "hex file"
r hhhh	- read memory at address hhhh
w hhhh dd	- write dd to memory at address hhhh
b ssss eeee	- dump memory block from ssss to eeee
f ssss eeee dd	- fill memory block from ssss to eeee with dd
g	- go - i.e. enter "run" mode
h	- halt - i.e. enter "halt" mode
j hhhh	- jump to address hhhh and reset into "run" mode
k hhhh	- jump to address hhhh and begin "single step" mode
n	- single step to next instruction
x cc	- read register "cc"
y cc dddd	- write dddd to register "cc"
z	- dump contents of all registers
t rate	- set clock tick to rate
?	- display CLI help
q	- quit the simulator

StackSim Class Diagram



The Core simulator emulates the internals of the StackCPU device. It provides interfaces for connecting to the Memory simulator (i.e. address, data, and control). It also provides debug interfaces for connecting with the UI simulator (i.e. control over operating modes, read/write of registers, etc).

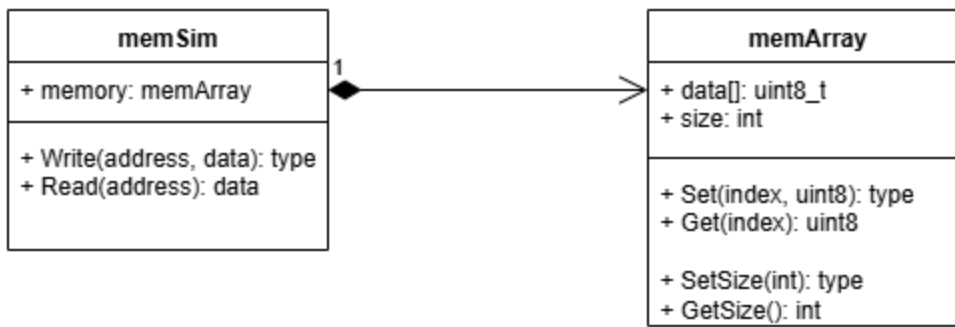
coreSim Class Diagram



Memory Simulator

The Memory simulator emulates the system memory of the StackCPU system. It provides interfaces for connecting to the Core simulator (i.e. address, data, and control). It also provides debug interfaces for connecting with the UI simulator (i.e. loading memory from binary files, saving memory to binary files, and read/write of locations in memory).

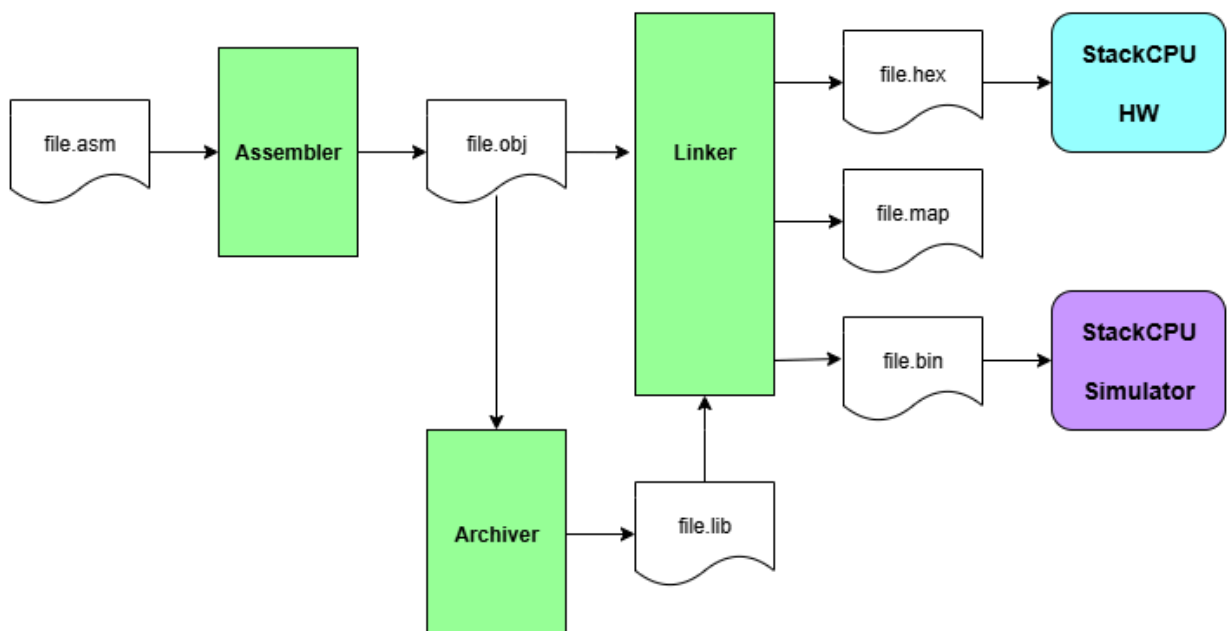
memSim Class Diagram



Assembler Tools

The StackCPU Assembler is a set of tools (implemented in C) for compiling StackCPU assembly source code into machine code that can be executed by a StackCPU simulator or a StackCPU HW implementation. This set of tools includes an assembler, an archiver, and a linker.

StackCPU Assembler Tools



Assembler

The Assembler tool compiles a StackCPU assembly source file into an object module. Various options allow some control over enabling of debug info, setting warning level, include search path, etc.

A special mode enables the direct output of an executable binary file from a single assembly source file using the "-b" option, but this not compatible with also outputting an object file. This is primarily intended as an early method of assembling a source file into a binary that can be executed on the simulator without the need for the linker.

Usage: stackasm [options] [source file]



options:

- | | |
|---------------|---|
| -g | - add debug info to object file |
| -D name[=val] | - define a symbol |
| -W n | - set warning level n |
| -I dir | - include directory search path |
| -b file.bin | - output binary executable (bypass object output) |
| -o file.obj | - object file output filename |
| -h | - display command arguments |
| -v | - verbose mode |
| -V | - display version |

Example: stackasm -g -I ./inc -o addloop.obj addloop.asm

Archiver

The Archiver tool collects object files into the specified library file. Various options enable control over whether to add, replace, or delete a file from the library.

Usage: stackar [options] [obj files]



options:

- | | |
|-------------|--|
| -a | - add object to specified library file |
| -d | - delete object from specified library |
| -r | - replace object in specified library |
| -o file.lib | - library file output filename |
| -h | - display command arguments |
| -v | - verbose mode |
| -V | - display version |

Example: stackar -a -o math.lib add.obj

Linker

The Linker tool combines several object modules (libraries and objects) producing an executable output file. Various options enable control over the linking process, and the additional output of a map file or a hex file in addition to the binary executable output. The map file is useful when debugging code in the simulator since it gives the address location of symbols within the executable program. The hex file is useful for programming an executable into a HW implementation.

Usage: `stackld [options] [lib files] [obj files]`



options:

<code>-C file.cfg</code>	- linker config file input
<code>-L dir</code>	- library directory search path
<code>-o file.bin</code>	- binary executable output filename
<code>-m file.map</code>	- map file output filename
<code>-x file.hex</code>	- hex file output filename
<code>-h</code>	- display command arguments
<code>-v</code>	- verbose mode
<code>-V</code>	- display version

Example: `stackld -m prog.map -o prog.bin math.lib addloop.obj`

Hardware

Logic Blocks

Fetch Logic

TBD

Instruction Decode Logic

TBD

ALU Logic

TBD

Control SM Logic

TBD

Implementations

Emulated Implementation

The Emulated Implementation is a simulation of the StackCPU running on a Pi Pico microcontroller. Basically this is a version of the Core Simulator noted below running on a Pi Pico.

FPGA Implementation

The FPGA Implementation is a HW design of the StackCPU device implemented on a TBD FPGA.

Discrete Implementation

The Discrete Implementation is a HW design of the StackCPU device implemented using discrete logic devices such as standard 7400 series TTL devices.

Tools Used

- DrawIO - <https://www.drawio.com/>
- Notepad++ - <https://notepad-plus-plus.org/>
- GVim - <https://www.vim.org/download.php>
- GitHub Desktop - <https://github.com/apps/desktop>
- Cygwin - <https://www.cygwin.com/>
 - gcc - from Cygwin
 - g++ - from Cygwin
 - make - from Cygwin
 - git - from Cygwin
- Custom tools
 - bindump - used to examine binary files
 - binedit - used to edit contents of binary file
 - bingen - used to generate binary files