Individual design and research project Stacks in ISAs

Andrés Antonio Bravo Orozco A01630783

Isaac Pérez Andrade

Thursday 4th of June 2020

A stack is a conceptual structure mainly used in programming and memory organization in computers. It refers to a set of elements of the same type, which all behave based on the principle of the Last In First Out (LIFO). This project will describe the stack and its functionality in modern ISAs as well as implementing a stack using System Verilog. This will allow a more complete understanding of the functionality of a stack and its implementations in modern technology.

A stack can perform two main operations: push and pop. These two instructions are carried on the topmost element of the set, which is the one most recently added to the stack. The push operations adds an element to the stack, making it the uppermost element in the set, while the pop operations removes an element from the top position. Most stacks also require a stack pointer, which keeps track of the uppermost element in the set; when a push operation is performed the top value (where the stack pointer must refer to) is incremented by one, while in a pop operations the top value is decremented by one. The stack can be dynamic and allow its size to change or bounded and therefore only capable of containing a certain number of elements, trying to add an element to an already full stack causes a stack overflow.

Hardware stacks are implemented for the purpose of memory allocation and access, using a fixed size. Stack based ISAs belong to an instruction set known as Zero Address Instructions. The name “zero-address” is given to this type of computer because of the absence of an address field in the instructions. This addresses are missing (except on PUSH and POP operations) since it is implied that the operations are performed on the two uppermost elements of the stack. An example of this is given in Fig. 1, where it is specified the instructions necessary to perform $X = (A + B) * (C + D)$. TOS refers to top of the stack.

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← (A + B)
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← (C + D)
MUL		TOS ← (C + D) * (A + B)
POP	X	M [X] ← TOS

Fig. 1. Stack operation example

In this example, we can see that the PUSH and POP operations still need an address field to specify which operands communicate with the stack, while ADD and MUL operations are implicitly performed on the last 2 elements of the stack, so no specifications of the addresses of these elements are needed.

The MIPS microprocessor is a register based ISA which works very differently to a stack based ISA. The main difference is in the instructions they receive. The way the

instructions are organized, as well as how much information needs to be specified in each instruction in order to perform the same operations varies between these two implementations. A direct comparison of this is given in Table 1, where the operation C = A + B is performed by the MIPS we designed (assuming A and B are in a memory locations 1 and 2 respectively) and a similar stack based ISA.

MIPS	Stack
LW R1, 1(R0)	PUSH A
LW R2, 2(R0)	PUSH B
ADD R3, R2, R1	ADD
SW R3, 3(R0)	POP C

Table 1. Comparison between Register ISA and Stack ISA

In this comparison, we can see that both the MIPS and the Stack need to load the values into a special memory (in the MIPS it's the registers while in the stack it's the stack itself) then perform the operation and give out the result, but the way the instructions are sent is very different. The MIPS has to perform different instructions in order to load the specific memory allocations for A and B into the registers for them to be summed, and the result be stored back in memory. The stack just needs to load both A and B values in the stack and then pop the result after it's summed. Another big difference is with respect to the addresses and therefore the length of the instructions. The instructions of stack ISAs are usually smaller because of these omitted addresses in the operations.

An example of a modern stack based ISA is the ZPU microprocessor, which is described as being the smallest 32 bits CPU. It is made by the Norwegian company Zylin AS to run supervisory code in electronic systems like field-programmable gate array (FPGA). This device uses a stack instead of registers since it can be made much smaller with this implementation, sacrificing speed in the process. The ZPU is designed to handle the miscellaneous tasks of a system that are mostly handled by software, for example, a user interface. The ZPU is very slow, but its small size helps to place any high-speed algorithm in the FPGA without requiring a lot of resources for itself. This ISA includes only immediate and direct addressing modes, since it performs all of its operations using the stack as memory. It uses a fixed encoding with an 8 bits opcode which includes the immediate instruction. Fig. 2 shows the basic instructions of the ZPU, most of these instructions interact with the stack in some way

Name	Binary	Description
BREAKPOINT	00000000	Halt the CPU and/or jump to the debugger.
IM_x	1xxxxxxx	Push or append a signed 7-bit immediate to the TOS.
STORESP_x	010xxxxx	Pop the TOS and store it into the stack at an offset from the top.
LOADSP_x	011xxxxx	Fetch from a value indexed in the stack and push it into the TOS.
EMULATE_x	001xxxxx	Emulate an instruction with code at vector x.
ADDSP_x	0001xxxx	Fetch from a value indexed in the stack and add the value to the TOS.
POPPC	00000100	Pop an address from the TOS and store it to the PC.
LOAD	00001000	Pop an address and push the loaded memory value to the TOS.
STORE	00001100	Store the NOS into the memory pointed-to by the TOS. Pop both.
PUSHSP	00000010	Push the current SP into the TOS.
POPSP	00001101	Pop the TOS and store it to the SP.
ADD	00000101	Integer addition of TOS and NOS.
AND	00000110	Bitwise AND of the TOS and NOS.
OR	00000111	Bitwise OR of the TOS and NOS.
NOT	00001001	Bitwise NOT of the TOS.
FLIP	00001010	Reverse the bit order of the TOS.
NOP	00001011	No-Operation. (Usually used for delay loops or tables of code.)

Fig. 2 ZPU Instruction Set

In the MIPS project, a stack could be used instead of the registers, allowing for shorter and easier instructions and, utilizing the same 32 bit instructions, allowing for more data to be utilized in the microprocessor data or shortening the number of bits required per instruction in order to have a more compact microprocessor.

The stack which was designed in System Verilog for this project works in a similar manner to what has been already described. This implementation of the stack contains 5 inputs consisting of clk, asyn_n_rst, push, pop, and data_in, and contains 3 outputs consisting of data_out, full, and empty, the last two being flags which signal if the stack is full or empty, respectively. In the System Verilog file two parameters were created in order to control the width of the data input and output, and the depth of the stack. These parameters are DATA_WIDTH and STACK_DEPTH. The push operation has priority over the pop operation, meaning that if both were to be turned on, only the push operation will be performed. Another specification of the design is that a push operation may only be made if the stack is not full, and the pop operation may only be performed if the stack is not empty.

The stack module I designed works with the help of the previously mentioned flags and a stack pointer which keeps track of the uppermost element of the stack. The flags are purely combinational; the empty flag will activate when the stack_pointer equals -1, while the full flag will activate when the stack_pointer equals the STACK_DEPTH. When a push operation is performed (considering that the stack is not full), the stack pointer will increment by one and incorporate the data in data_in into the stack in the position of the stack_pointer. When a pop operation is performed (considering it is not empty) the data_out will be updated with the last element introduced to the stack, afterwards the data in this position is set to 0 and the stack_pointer is decremented. The empty flag activates at stack_pointer = -1 to distinguish between the two situations in which the stack_pointer could be 0: stack pointer being at 0 with data on register 0, and stack pointer being at 0 with no data in register 0.

In Fig. 3 there is a screenshot of the ModelSim simulation, where DATA_WIDTH is set to 6 and STACK_DEPTH set to 16. Since the push operation has priority over the pop operation, pop is always set to 1 and push varies between 1 and 0 to enable the pop operation. The changes in the stack are shown on the positive edges of the clock. Fig. 4 contains a screenshot of the generated Register-Transfer Level (RTL) in Quartus, with the same parameters mentioned above.

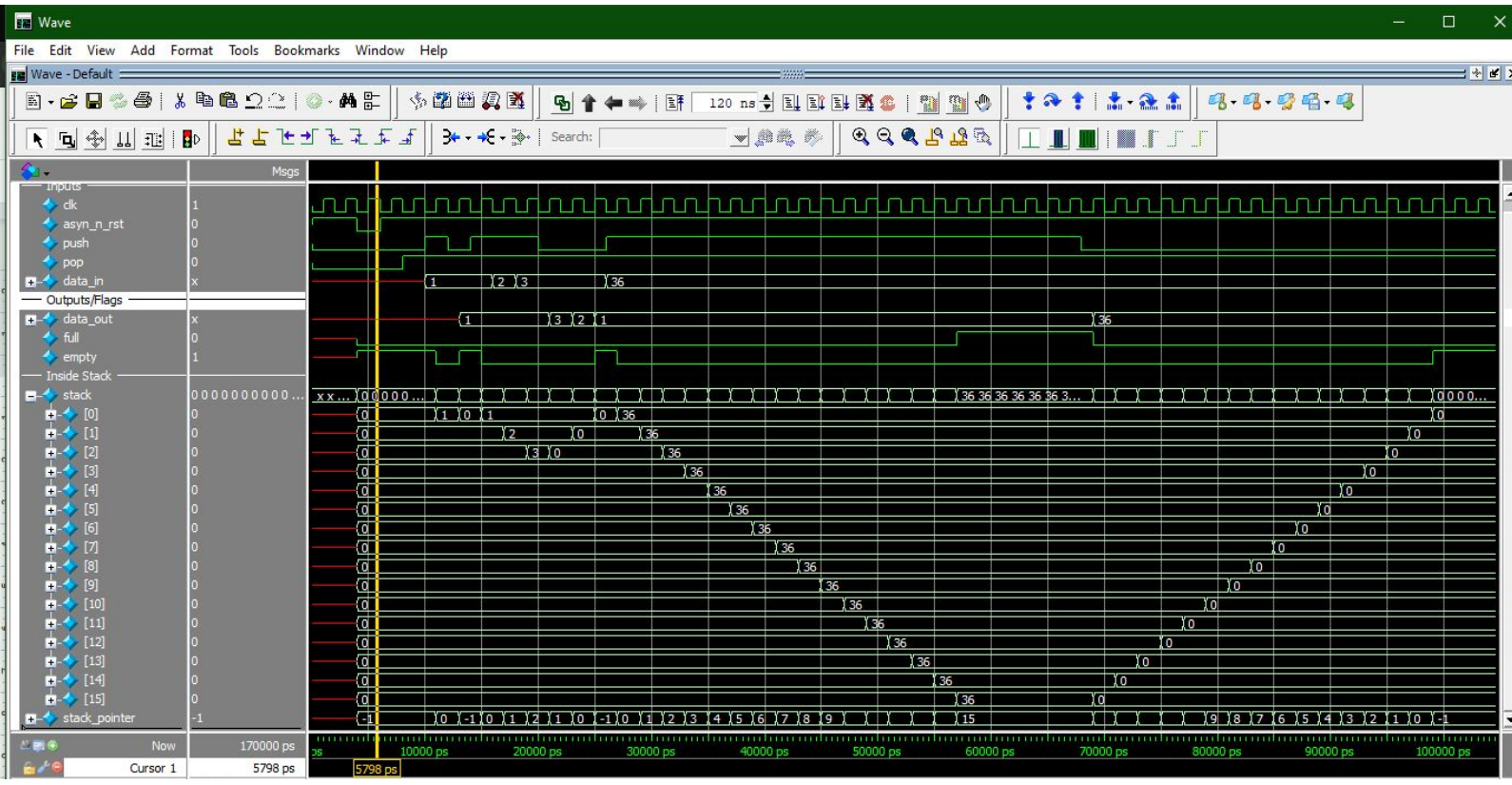

Fig. 3 Screenshot of ModelSim simulation.

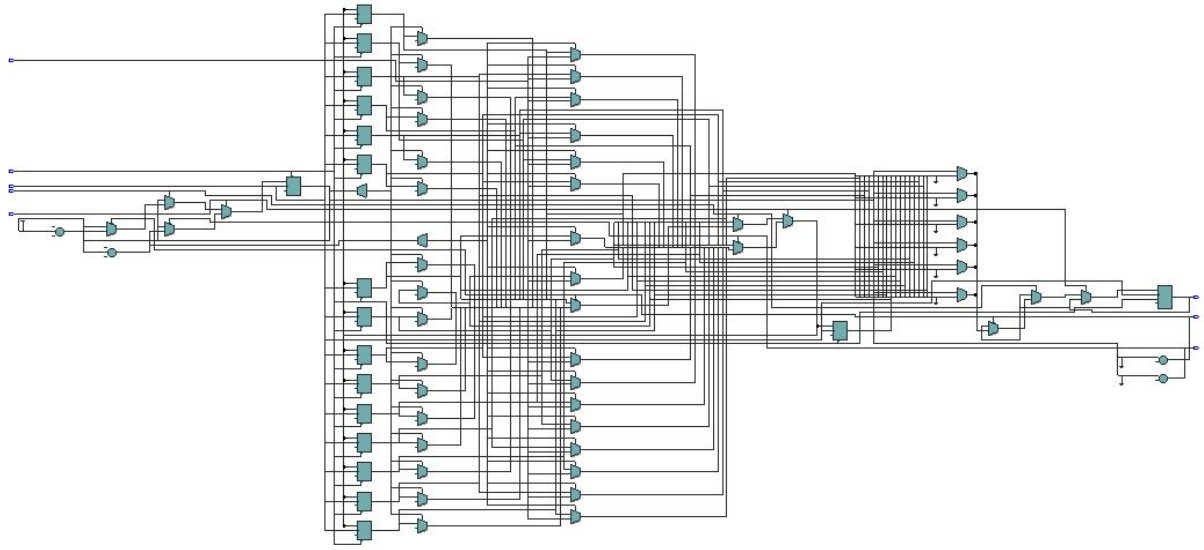

Fig 4. Screenshot of the generated Register-Transfer Level (RTL) in Quartus.

This project allowed for a deeper understanding of stacks, their hardware implementations in ISAs and their functionality. Before this project, I had only heard of stacks but never actually understood or researched them, therefore the process of investigating about these hardware modules was very new and enriching. I consider the stack, and more importantly the concept of the stack and how it works, to be fundamental in understanding more about the different ways data is allocated and processed in the devices we interact with, directly or indirectly, everyday.

References

"Stack.". techopedia. https://www.techopedia.com/definition/9523/stack (accessed May 14, 2020).

M. Panda. "Three, two, one and zero address instruction. programming1011. https://www.programming1011.com/2019/04/explain-three-two-one-and-zero-address_27.html (accessed May 20, 2020).

"Instruction formats". IARE. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=22&ved=2ahUKEwixqoeK5b7pAhUIXa0KHewIDzIQFjAVegQIARAB&url=https%3A%2F%2Fwww.iare.ac.in%2Fsites%2Fdefault%2Ffiles%2FPPT%2FCO%2520Lecture%2520Notes.pdf&usg=AOvVaw0YMjE-M2IUxLJbD-g6jGsM (accessed May 20, 2020).

"Instruction Set Architecutre ISA". Kent. http://www.cs.kent.edu/~durand/CS0/Notes/Chapter05/isa.html (accessed May 20, 2020).

"A New Old Processor". h4ck3r.net. https://h4ck3r.net/2010/03/02/a-new-old-processor/ (accessed May 20, 2020).

"Zog - A ZPU processor core". parallax. https://forums.parallax.com/discussion/119711/zog-a-zpu-processor-core-for-the-prop-gnu-c-c-and-fortran-now-replaces-s (accessed May 20, 2020).

H. Øyvind. "ZPU - the worlds smallest 32 bit CPU with GCC toolchain". opencores. https://opencores.org/projects/zpu (accessed May 20, 2020).

"ZPUFlex" Retro Ramblings. http://retroramblings.net/?page_id=627 (accessed May 20, 2020).