WIKIPEDIA

Stack machine

In computer science, computer engineering and programming language implementations, a **stack machine** is a computer processor or a virtual machine in which the primary interaction is moving short-lived temporary values to and from a push down stack. In the case of a hardware processor, a hardware stack is used. The use of a stack significantly reduces the required number of processor registers. Stack machines extend push-down automata with additional load/store operations or multiple stacks and hence are Turing-complete.

Contents

Design
- Stack storage

History and implementations
- Commercial stack machines
- Virtual stack machines
- Hybrid machines
- Computers using call stacks and stack frames

Comparison with register machines
- Instructions
- Temporary / local values
- Common subexpressions
- Pipelining
- Out-of-order execution
- Hides a faster register machine inside
- Interrupts
- Interpreters

See also

References

External links

Design

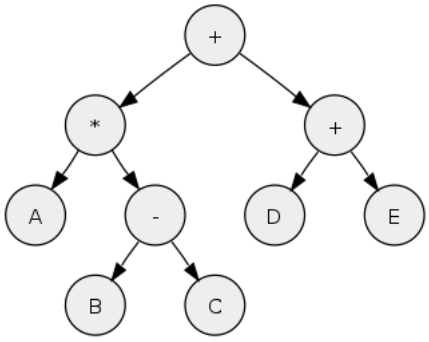
Most or all stack machine instructions assume that operands will be from the stack, and results placed in the stack. The stack easily holds more than two inputs or more than one result, so a rich set of operations can be computed. In stack machine code (sometimes called p-code), instructions will frequently have only an opcode commanding an operation, with no additional fields identifying a constant, register or memory cell, known as a **zero address format**.[1] This greatly simplifies instruction decoding. Branches, load immediates, and load/store instructions require an argument field, but stack machines often arrange that the frequent cases of these still fit together with the opcode into a compact group of bits. The selection of operands from prior results is done implicitly by ordering the instructions. Some stack machine instruction sets are intended for interpretive execution of a virtual machine, rather than driving hardware directly.

Integer constant operands are pushed by `Push` or `Load Immediate` instructions. Memory is often accessed by separate `Load` or `Store` instructions containing a memory address or calculating the address from values in the stack. All practical stack machines have variants of the load–store opcodes for accessing local variables and formal parameters without explicit address calculations. This can be by offsets from the current top-of-stack address, or by offsets from a stable frame-base register.

The instruction set carries out most ALU actions with postfix (reverse Polish notation) operations that work only on the expression stack, not on data registers or main memory cells. This can be very convenient for executing high-level languages, because most arithmetic expressions can be easily translated into postfix notation.

For example, consider the expression *A**(*B*-*C*)+(*D*+*E*), written in reverse Polish notation as *A B C* - * *D E* + +. Compiling and running this on a simple imaginary stack machine would take the form:

```
                # stack contents (leftmost = top = most recent):
push A          #           A
push B          #     B     A
push C          # C   B     A
subtract        #     B-C   A
multiply        #           A*(B-C)
push D          #     D     A*(B-C)
push E          # E   D     A*(B-C)
add             #     D+E   A*(B-C)
add             #           A*(B-C)+(D+E)
```



Binary syntax tree for the expression *A**(*B*-*C*) + (*D*+*E*)

The arithmetic operations 'subtract', 'multiply', and 'add' act on the two topmost operands of the stack. The computer takes both operands from the topmost (most recent) values of the stack. The computer replaces those two values with the calculated difference, sum, or product. In other words the instruction's operands are "popped" off the stack, and its result(s) are then "pushed" back onto the stack, ready for the next instruction.

Stack machines may have their expression stack and their call-return stack separated or as one integrated structure. If they are separated, the instructions of the stack machine can be pipelined with fewer interactions and less design complexity, so it will usually run faster.

Optimisation of compiled stack code is quite possible. Back-end optimisation of compiler output has been demonstrated to significantly improve code,[2][3] and potentially performance, whilst global optimisation within the compiler itself achieves further gains.[4]

Stack storage

Some stack machines have a stack of limited size, implemented as a register file. The ALU will access this with an index. A large register file uses a lot of transistors and hence this method is only suitable for small systems. A few machines have both an expression stack in memory and a separate register stack. In this case, software, or an interrupt may move data between them. Some machines have a stack of unlimited size, implemented as an array in RAM, which is cached by some number "top of stack" address registers to reduce memory access. Except for explicit "load from memory" instructions, the order of operand usage is identical with the order of the operands in the data stack, so excellent prefetching can be accomplished easily.

Consider `X+1`. It compiles to `Load X`; `Load 1`; `Add`. With a stack stored completely in RAM, this does implicit writes and reads of the in-memory stack:

- Load X, push to memory
- Load 1, push to memory
- Pop 2 values from memory, add, and push result to memory

for a total of 5 data cache references.

The next step up from this is a stack machine or interpreter with a single top-of-stack register. The above code then does:

- Load X into empty TOS register (if hardware machine) **or** Push TOS register to memory, Load X into TOS register (if interpreter)
- Push TOS register to memory, Load 1 into TOS register
- Pop left operand from memory, add to TOS register and leave it there

for a total of 5 data cache references, worst-case. Generally, interpreters don't track emptiness, because they don't have to—anything below the stack pointer is a non-empty value, and the TOS cache register is always kept hot. Typical Java interpreters do not buffer the top-of-stack this way, however, because the program and stack have a mix of short and wide data values.

If the hardwired stack machine has 2 or more top-stack registers, or a register file, then all memory access is avoided in this example and there is only 1 data cache cycle.

History and implementations

Description of such a method requiring only two values at a time to be held in registers, with a limited set of pre-defined operands that were able to be extended by definition of further operands, functions and subroutines, was first provided at conference by Robert S. Barton in 1961.[5][6]

Commercial stack machines

Examples of stack instruction sets directly executed in hardware include

- the Z4 (1945) computer by Konrad Zuse.[7][8]
- the Burroughs large systems architecture (since 1961)
- the English Electric KDF9 machine. First delivered in 1964, the KDF9 had a 19-level deep pushdown stack of arithmetic registers, and a 17-level deep stack for subroutine return addresses
- the Collins Radio Collins Adaptive Processing System minicomputer (CAPS, since 1969) and Rockwell Collins Advanced Architecture Microprocessor (AAMP, since 1981).[9]
- the Xerox Dandelion introduced 27 April 1981, utilized a stack machine architecture to save memory.[10][11]
- the UCSD Pascal p-machine (as the Pascal MicroEngine and many others) supported a complete student programming environment on early 8-bit microprocessors with poor instruction sets and little RAM, by compiling to a virtual stack machine.
- MU5 and ICL 2900 Series. Hybrid stack and accumulator machines. The accumulator register buffered the memory stack's top data value. Variants of load and store opcodes controlled when that register was spilled to the memory stack or reloaded from there.
- HP 3000 (Classic, not PA-RISC)
- Tandem Computers T/16. Like HP 3000, except that compilers, not microcode, controlled when the register stack spilled to the memory stack or was refilled from the memory stack.

- the Atmel MARC4 microcontroller[12]
- Several "Forth chips"[13] such as the RTX2000, the RTX2010, the F21[14] and the PSC1000[15][16]
- The Setun Ternary computer performed balanced ternary using a stack.
- The 4stack processor by Bernd Paysan has four stacks.[17]
- Patriot Scientific's Ignite stack machine designed by Charles H. Moore holds a leading *functional density* benchmark.
- Saab Ericsson Space Thor radiation hardened microprocessor[18]
- Inmos transputers.
- ZPU A physically-small CPU designed to supervise FPGA systems.[19]
- The F18A architecture of the 144-processor GA144 chip from GreenArrays, Inc.[20][21][22]
- Some technical handheld calculators use reverse Polish notation in their keyboard interface, instead of having parenthesis keys. This is a form of stack machine. The Plus key relies on its two operands already being at the correct topmost positions of the user-visible stack.

Virtual stack machines

Examples of virtual stack machines interpreted in software:

- the Whetstone ALGOL 60 interpretive code,[23] on which some features of the Burroughs B6500 were based
- the UCSD Pascal p-machine; which closely resembled Burroughs
- the Niklaus Wirth p-code machine
- Smalltalk
- the Java virtual machine instruction set (note that only the abstract instruction set is stack based, HotSpot, the Sun Java Virtual Machine for instance, does not implement the actual interpreter in software, but as handwritten assembly stubs)
- the WebAssembly bytecode
- the Virtual Execution System (VES) for the Common Intermediate Language (CIL) instruction set of the .NET Framework (ECMA 335)
- the Forth programming language, especially the integral virtual machine
- Adobe's PostScript
- Parakeet programming language
- Sun Microsystems' SwapDrop programming language for Sun Ray smartcard identification
- Adobe's ActionScript Virtual Machine 2 (AVM2)
- Ethereum's EVM
- the CPython bytecode interpreter
- the Ruby YARV bytecode interpreter
- the Rubinius virtual machine
- the bs (programming language) in Unix uses a virtual stack machine to process commands, after first transposing provided input language form, into reverse-polish notation
- the Lua (programming language) C API

Hybrid machines

Pure stack machines are quite inefficient for procedures which access multiple fields from the same object. The stack machine code must reload the object pointer for each pointer+offset calculation. A common fix for this is to add some register-machine features to the stack machine: a

visible register file dedicated to holding addresses, and register-style instructions for doing loads and simple address calculations. It is uncommon to have the registers be fully general purpose, because then there is no strong reason to have an expression stack and postfix instructions.

Another common hybrid is to start with a register machine architecture, and add another memory address mode which emulates the push or pop operations of stack machines: 'memaddress = reg; reg += instr.displ'. This was first used in DEC's PDP-11 minicomputer. This feature was carried forward in VAX computers and in Motorola 6800 and M68000 microprocessors. This allowed the use of simpler stack methods in early compilers. It also efficiently supported virtual machines using stack interpreters or threaded code. However, this feature did not help the register machine's own code to become as compact as pure stack machine code. Also, the execution speed was less than when compiling well to the register architecture. It is faster to change the top-of-stack pointer only occasionally (once per call or return) rather than constantly stepping it up and down throughout each program statement, and it is even faster to avoid memory references entirely.

More recently, so-called second-generation stack machines have adopted a dedicated collection of registers to serve as address registers, off-loading the task of memory addressing from the data stack. For example, MuP21 relies on a register called "A", while the more recent GreenArrays processors relies on two registers: A and B.[21]

The Intel x86 family of microprocessors have a register-style (accumulator) instruction set for most operations, but use stack instructions for its x87, Intel 8087 floating point arithmetic, dating back to the iAPX87 (8087) coprocessor for the 8086 and 8088. That is, there are no programmer-accessible floating point registers, but only an 80-bit wide, 8-level deep stack. The x87 relies heavily on the x86 CPU for assistance in performing its operations.

Computers using call stacks and stack frames

Most current computers (of any instruction set style) and most compilers use a large call-return stack in memory to organize the short-lived local variables and return links for all currently active procedures or functions. Each nested call creates a new **stack frame** in memory, which persists until that call completes. This call-return stack may be entirely managed by the hardware via specialized address registers and special address modes in the instructions. Or it may be merely a set of conventions followed by the compilers, using generic registers and register+offset address modes. Or it may be something in between.

Since this technique is now nearly universal, even on register machines, it is not helpful to refer to all these machines as stack machines. That term is commonly reserved for machines which also use an expression stack and stack-only arithmetic instructions to evaluate the pieces of a single statement.

Computers commonly provide direct, efficient access to the program's global variables and to the local variables of only the current innermost procedure or function, the topmost stack frame. 'Up level' addressing of the contents of callers' stack frames is usually not needed and not supported as directly by the hardware. If needed, compilers support this by passing in frame pointers as additional, hidden parameters.

Some Burroughs stack machines do support up-level refs directly in the hardware, with specialized address modes and a special 'display' register file holding the frame addresses of all outer scopes. Currently, only MCST Elbrus has done this in hardware. When Niklaus Wirth developed the first Pascal compiler for the CDC 6000, he found that it was faster overall to pass in the frame pointers as a chain, rather than constantly updating complete arrays of frame pointers. This software method also adds no overhead for common languages like C which lack up-level refs.

The same Burroughs machines also supported nesting of tasks or threads. The task and its creator share the stack frames that existed at the time of task creation, but not the creator's subsequent frames nor the task's own frames. This was supported by a cactus stack, whose layout diagram resembled the trunk and arms of a Saguaro cactus. Each task had its own memory segment holding its stack and the frames that it owns. The base of this stack is linked to the middle of its creator's stack. In machines with a conventional flat address space, the creator stack and task stacks would be separate heap objects in one heap.

In some programming languages, the outer-scope data environments are not always nested in time. These languages organize their procedure 'activation records' as separate heap objects rather than as stack frames appended to a linear stack.

In simple languages like Forth that lack local variables and naming of parameters, stack frames would contain nothing more than return branch addresses and frame management overhead. So their return stack holds bare return addresses rather than frames. The return stack is separate from the data value stack, to improve the flow of call setup and returns.

Comparison with register machines

Stack machines are often compared to register machines, which hold values in an array of registers. Register machines may store stack-like structures in this array, but a register machine has instructions which circumvent the stack interface. Register machines routinely outperform stack machines,[24] and stack machines have remained a niche player in hardware systems. But stack machines are often used in implementing virtual machines because of their simplicity and ease of implementation.[25]

Instructions

Stack machines have higher code density. In contrast to common stack machine instructions which can easily fit in 6 bits or less, register machines require two or three register-number fields per ALU instruction to select operands; the densest register machines average about 16 bits per instruction plus the operands. Register machines also use a wider offset field for load-store opcodes. A stack machine's compact code naturally fits more instructions in cache, and therefore could achieve better cache efficiency, reducing memory costs or permitting faster memory systems for a given cost. In addition, most stack-machine instruction is very simple, made from only one opcode field or one operand field. Thus, stack-machines require very little electronic resources to decode each instruction.

A program has to execute more instructions when compiled to a stack machine than when compiled to a register machine or memory-to-memory machine. Every variable load or constant requires its own separate Load instruction, instead of being bundled within the instruction which uses that value. The separated instructions may be simple and faster running, but the total instruction count is still higher.

Most register interpreters specify their registers by number. But a host machine's registers can't be accessed in an indexed array, so a memory array is allotted for virtual registers. Therefore, the instructions of a register interpreter must use memory for passing generated data to the next instruction. This forces register interpreters to be much slower on microprocessors made with a fine process rule (i.e. faster transistors without improving circuit speeds, such as the Haswell x86). These require several clocks for memory access, but only one clock for register access. In the case of a stack machine with a data forwarding circuit instead of a register file, stack interpreters can allot the host machine's registers for the top several operands of the stack instead of the host machine's memory

In a stack machine, the operands used in the instructions are always at a known offset (set in the stack pointer), from a fixed location (the bottom of the stack, which in a hardware design might always be at memory location zero), saving precious in-cache or in-CPU storage from being used to store quite so many memory addresses or index numbers. This may preserve such registers and cache for use in non-flow computation.

Temporary / local values

Some in the industry believe that stack machines execute more data cache cycles for temporary values and local variables than do register machines.[26]

On stack machines, temporary values often get spilled into memory, whereas on machines with many registers these temps usually remain in registers. (However, these values often need to be spilled into "activation frames" at the end of a procedure's definition, basic block, or at the very least, into a memory buffer during interrupt processing). Values spilled to memory add more cache cycles. This spilling effect depends on the number of hidden registers used to buffer top-of-stack values, upon the frequency of nested procedure calls, and upon host computer interrupt processing rates.

On register machines using optimizing compilers, it is very common for the most-used local variables to remain in registers rather than in stack frame memory cells. This eliminates most data cache cycles for reading and writing those values. The development of "stack scheduling" for performing live-variable analysis, and thus retaining key variables on the stack for extended periods, helps this concern.

On the other hand, register machines must spill many of their registers to memory across nested procedure calls. The decision of which registers to spill, and when, is made statically at compile time rather than on the dynamic depth of the calls. This can lead to more data cache traffic than in an advanced stack machine implementation.

Common subexpressions

In register machines, a common subexpression (a subexpression which is used multiple times with the same result value) can be evaluated just once and its result saved in a fast register. The subsequent reuses have no time or code cost, just a register reference. This optimization speeds simple expressions (for example, loading variable X or pointer P) as well as less-common complex expressions.

With stack machines, in contrast, results can be stored in one of two ways. Firstly, results can be stored using a temporary variable in memory. Storing and subsequent retrievals cost additional instructions and additional data cache cycles. Doing this is only a win if the subexpression computation costs more in time than fetching from memory, which in most stack CPUs, almost always is the case. It is never worthwhile for simple variables and pointer fetches, because those already have the same cost of one data cache cycle per access. It is only marginally worthwhile for expressions such as `X+1`. These simpler expressions make up the majority of redundant, optimizable expressions in programs written in non-concatenative languages. An optimizing compiler can only win on redundancies that the programmer could have avoided in the source code.

The second way leaves a computed value on the data stack, duplicating it as needed. This uses operations to copy stack entries. The stack must be depth shallow enough for the CPU's available copy instructions. Hand-written stack code often uses this approach, and achieves speeds like

general-purpose register machines.[27][8] Unfortunately, algorithms for optimal "stack scheduling" are not in wide use by programming languages.

Pipelining

In modern machines, the time to fetch a variable from the data cache is often several times longer than the time needed for basic ALU operations. A program runs faster without stalls if its memory loads can be started several cycles before the instruction that needs that variable. Complex machines can do this with a deep pipeline and "out-of-order execution" that examines and runs many instructions at once. Register machines can even do this with much simpler "in-order" hardware, a shallow pipeline, and slightly smarter compilers. The load step becomes a separate instruction, and that instruction is statically scheduled much earlier in the code sequence. The compiler puts independent steps in between.

Scheduling memory accesses requires explicit, spare registers. It is not possible on stack machines without exposing some aspect of the micro-architecture to the programmer. For the expression A B -, B must be evaluated and pushed immediately prior to the Minus step. Without stack permutation or hardware multithreading, relatively little useful code can be put in between while waiting for the Load B to finish. Stack machines can work around the memory delay by either having a deep out-of-order execution pipeline covering many instructions at once, or more likely, they can permute the stack such that they can work on other workloads while the load completes, or they can interlace the execution of different program threads, as in the Unisys A9 system.[28] Today's increasingly parallel computational loads suggests, however, this might not be the disadvantage it's been made out to be in the past.

Stack machines can omit the operand fetching stage of a register machine.[27] For example, in the Java Optimized Processor (JOP) microprocessor the top 2 operands of stack directly enter a data forwarding circuit that is faster than the register file.[29]

Out-of-order execution

The Tomasulo algorithm finds instruction-level parallelism by issuing instructions as their data becomes available. Conceptually, the addresses of positions in a stack are no different than the register indexes of a register file. This view permits the out-of-order execution of the Tomasulo algorithm to be used with stack machines.

Out-of-order execution in stack machines seems to reduce or avoid many theoretical and practical difficulties.[30] The cited research shows that such a stack machine can exploit instruction-level parallelism, and the resulting hardware must cache data for the instructions. Such machines effectively bypass most memory accesses to the stack. The result achieves throughput (instructions per clock) comparable to RISC register machines, with much higher code densities (because operand addresses are implicit).

One issue brought up in the research was that it takes about 1.88 stack-machine instructions to do the work of a register machine's RISC instruction. Competitive out-of-order stack machines therefore require about twice as many electronic resources to track instructions ("issue stations"). This might be compensated by savings in instruction cache and memory and instruction decoding circuits.

Hides a faster register machine inside

Some simple stack machines have a chip design which is fully customized all the way down to the level of individual registers. The top of stack address register and the N top of stack data buffers are built from separate individual register circuits, with separate adders and ad hoc connections.

However, most stack machines are built from larger circuit components where the N data buffers are stored together within a register file and share read/write buses. The decoded stack instructions are mapped into one or more sequential actions on that hidden register file. Loads and ALU ops act on a few topmost registers, and implicit spills and fills act on the bottommost registers. The decoder allows the instruction stream to be compact. But if the code stream instead had explicit register-select fields which directly manipulated the underlying register file, the compiler could make better use of all registers and the program would run faster.

Microprogrammed stack machines are an example of this. The inner microcode engine is some kind of RISC-like register machine or a VLIW-like machine using multiple register files. When controlled directly by task-specific microcode, that engine gets much more work completed per cycle than when controlled indirectly by equivalent stack code for that same task.

The object code translators for the HP 3000 and Tandem T/16 are another example.[31][32] They translated stack code sequences into equivalent sequences of RISC code. Minor 'local' optimizations removed much of the overhead of a stack architecture. Spare registers were used to factor out repeated address calculations. The translated code still retained plenty of emulation overhead from the mismatch between original and target machines. Despite that burden, the cycle efficiency of the translated code matched the cycle efficiency of the original stack code. And when the source code was recompiled directly to the register machine via optimizing compilers, the efficiency doubled. This shows that the stack architecture and its non-optimizing compilers were wasting over half of the power of the underlying hardware.

Register files are good tools for computing because they have high bandwidth and very low latency, compared to memory references via data caches. In a simple machine, the register file allows reading two independent registers and writing of a third, all in one ALU cycle with one-cycle or less latency. Whereas the corresponding data cache can start only one read or one write (not both) per cycle, and the read typically has a latency of two ALU cycles. That's one third of the throughput at twice the pipeline delay. In a complex machine like Athlon that completes two or more instructions per cycle, the register file allows reading of four or more independent registers and writing of two others, all in one ALU cycle with one-cycle latency. Whereas the corresponding dual-ported data cache can start only two reads or writes per cycle, with multiple cycles of latency. Again, that's one third of the throughput of registers. It is very expensive to build a cache with additional ports.

Since a stack is a component of most software programs, even when the software used is not strictly a stack machine, a hardware stack machine might more closely mimic the inner workings of its programs. Processor registers have a high thermal cost, and a stack machine might claim higher energy efficiency.[20]

Interrupts

Responding to an interrupt involves saving the registers to a stack, and then branching to the interrupt handler code. Often stack machines respond more quickly to interrupts, because most parameters are already on a stack and there is no need to push them there. Some register machines deal with this by having multiple register files that can be instantly swapped[33] but this increases costs and slows down the register file.

Interpreters

Interpreters for virtual stack machines are easier to build than interpreters for register machines; the logic for handling memory address modes is in just one place rather than repeated in many instructions. Stack machines also tend to have fewer variations of an opcode; one generalized opcode will handle both frequent cases and obscure corner cases of memory references or function call setup. (But code density is often improved by adding short and long forms for the same operation.)

Interpreters for virtual stack machines are often slower than interpreters for other styles of virtual machine.[34] This slowdown is worst when running on host machines with deep execution pipelines, such as current x86 chips.

In some interpreters, the interpreter must execute a N-way switch jump to decode the next opcode and branch to its steps for that particular opcode. Another method for selecting opcodes is threaded code. The host machine's prefetch mechanisms are unable to predict and fetch the target of that indexed or indirect jump. So the host machine's execution pipeline must restart each time the hosted interpreter decodes another virtual instruction. This happens more often for virtual stack machines than for other styles of virtual machine.[35]

One example is the Java programming language. Its canonical virtual machine is specified as an 8-bit stack machine. However, the Dalvik virtual machine for Java used on Android smartphones is a 16-bit virtual-register machine - a choice made for efficiency reasons. Arithmetic instructions directly fetch or store local variables via 4-bit (or larger) instruction fields.[36] Similarly version 5.0 of Lua replaced its virtual stack machine with a faster virtual register machine.[37][38]

Since Java virtual machine became popular, microprocessors have employed advanced branch predictors for indirect jumps.[39] This advance avoids most of pipeline restarts from N-way jumps and eliminates much of the instruction count costs that affect stack interpreters.

See also

- Stack-oriented programming language
- Concatenative programming language
- Comparison of application virtual machines
- SECD machine
- Accumulator machine
- Belt machine
- Random-access machine

References

1. Beard, Bob (Autumn 1997). "The KDF9 Computer - 30 Years On" (http://www.cs.man.ac.uk/CCS/res/res18.htm#c). *Computer RESURRECTION*.
2. Koopman, Jr., Philip John (1994). "A Preliminary Exploration of Optimized Stack Code Generation" (http://www.ece.cmu.edu/~koopman/stack_compiler/stack_co.pdf) (PDF). *Journal of Forth Applications and Research*. **6** (3).
3. Bailey, Chris (2000). "Inter-Boundary Scheduling of Stack Operands: A preliminary Study" (http://www.complang.tuwien.ac.at/anton/euroforth/ef00/bailey00.pdf) (PDF). *Proceedings of Euroforth 2000 Conference*.
4. Shannon, Mark; Bailey, Chris (2006). "Global Stack Allocation: Register Allocation for Stack Machines" (http://www.complang.tuwien.ac.at/anton/euroforth2006/papers/shannon.pdf) (PDF). *Proceedings of Euroforth Conference 2006*.

5. Barton, Robert S. (1961). "A new approach to the functional design of a digital computer" (https://dl.acm.org/doi/10.1145/1460690.1460736). *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. 1961 Western Joint IRE-AIEE-ACM Computer Conference. pp. 393–396. doi:10.1145/1460690.1460736 (https://doi.org/10.1145%2F1460690.1460736). ISBN 978-1-45037872-7. S2CID 29044652 (https://api.semanticscholar.org/CorpusID:29044652).
6. Barton, Robert S. (1987). "A new approach to the functional design of a digital computer" (http://doi.ieeecomputersociety.org/10.1109/MAHC.1987.10002). *IEEE Annals of the History of Computing*. **9**: 11–15. doi:10.1109/MAHC.1987.10002 (https://doi.org/10.1109%2FMAHC.1987.10002).
7. Blaauw, Gerrit Anne; Brooks, Jr., Frederick Phillips (1997). *Computer architecture: Concepts and evolution*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc.
8. LaForest, Charles Eric (April 2007). "2.1 Lukasiewicz and the First Generation: 2.1.2 Germany: Konrad Zuse (1910–1995); 2.2 The First Generation of Stack Computers: 2.2.1 Zuse Z4". *Second-Generation Stack Computer Architecture* (http://fpgacpu.ca/publications/Second-Generation_Stack_Computer_Architecture.pdf) (PDF) (thesis). Waterloo, Canada: University of Waterloo. p. 8, 11, etc. Archived (https://web.archive.org/web/20220120155616/http://fpgacpu.ca/publications/Second-Generation_Stack_Computer_Architecture.pdf) (PDF) from the original on 2022-01-20. Retrieved 2022-07-02. (178 pages) [1] (https://web.archive.org/web/20110718112702/http://www.eecg.utoronto.ca/~laforest/Second-Generation_Stack_Computer_Architecture.pdf)
9. Greve, David A.; Wilding, Matthew M. (1998-01-12). "The World's First Java Processor" (http://hokiepokie.org/docs/EETimes.ps). *Electronic Engineering Times*.
10. "Mesa Processor Principles of Operation" (http://www.digibarn.com/friends/alanfreier/princops/00yTableOfContents.html). *DigiBarn Computer Museum*. Xerox. Retrieved 2019-12-23.
11. "DigiBarn: The Xerox Star 8010 "Dandelion" " (http://www.digibarn.com/collections/systems/xerox-8010/index.html). DigiBarn Computer Museum. Retrieved 2019-12-23.
12. *MARC4 4-bit Microcontrollers Programmer's Guide* (https://en.wikichip.org/w/images/4/44/MARC4_4-bit_Microcontrollers_Programmer%27s_Guide.pdf) (PDF). Atmel.
13. "Forth chips" (https://web.archive.org/web/20060215200605/http://www.colorforth.com/chips.html). *Colorforth.com*. Archived from the original (http://www.colorforth.com/chips.html) on 2006-02-15. Retrieved 2017-10-08.
14. "F21 Microprocessor Overview" (http://www.ultratechnology.com/f21.html). *Ultratechnology.com*. Retrieved 2017-10-08.
15. "ForthFreak wiki" (https://github.com/ForthHub/ForthFreak). *GitHub.com*. 2017-08-25. Retrieved 2017-10-08.
16. "A Java chip available -- now!" (https://www.developer.com/guides/a-java-chip-available-now/). *Developer.com*. 1999-04-08. Retrieved 2022-07-07.
17. "4stack Processor" (http://bernd-paysan.de/4stack.html). *bernd-paysan.de*. Retrieved 2017-10-08.
18. "Porting the GNU C Compiler to the Thor Microprocessor" (https://web.archive.org/web/20110820085702/http://lundqvist.dyndns.org/Publications/thesis95/ThorGCC.pdf) (PDF). 1995-12-04. Archived from the original (http://lundqvist.dyndns.org/Publications/thesis95/ThorGCC.pdf) (PDF) on 2011-08-20. Retrieved 2011-03-30.
19. "ZPU - the world's smallest 32-bit CPU with a GCC tool-chain: Overview" (http://opencores.org/project,zpu). opencores.org. Retrieved 2015-02-07.
20. "Documents" (https://www.greenarraychips.com/home/documents/index.php#F18A). *GreenArrays, Inc*. F18A Technology. Retrieved 2022-07-07.
21. "colorForth Instructions" (https://web.archive.org/web/20160310112802/http://colorforth.com/inst.htm). *Colorforth.com*. Archived from the original (http://www.colorforth.com/inst.htm) on 2016-03-10. Retrieved 2017-10-08. (Instruction set of the F18A cores, named colorForth for historical reasons.)

22. "GreenArrays, Inc" (http://www.greenarraychips.com/). *Greenarraychips.com*. Retrieved 2017-10-08.
23. Randell, Brian; Russell, Lawford John (1964). *Algol 60 Implementation* (http://www.softwarepreservation.org/projects/ALGOL/book/Randell_ALGOL_60_Implementation_1964.pdf) (PDF). London, UK: Academic Press. ISBN 0-12-578150-4.
24. Shi, Yunhe; Gregg, David; Beatty, Andrew; Ertl, M. Anton (2005). "Virtual machine showdown: stack versus registers". *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments - VEE '05*: 153. doi:10.1145/1064979.1065001 (https://doi.org/10.1145%2F1064979.1065001). S2CID 811512 (https://api.semanticscholar.org/CorpusID:811512).
25. Hyde, Randall (2004). *Write Great Code, Vol. 2: Thinking Low-Level, Writing High-Level* (https://www.google.com/books/edition/Write_Great_Code_Vol_2/mM58oD4LATUC?hl=en&gbpv=1&dq=stack%20machines%20simplicity&pg=PA391&printsec=frontcover&bsq=stack%20machines%20simplicity). Vol. 2. No Starch Press. p. 391. ISBN 978-1-59327-065-0. Retrieved 2021-06-30.
26. "Computer Architecture: A Quantitative Approach", John L. Hennessy, David Andrew Patterson; See the discussion of stack machines.
27. Koopman, Jr., Philip John. "Stack Computers: the new wave" (http://www.ece.cmu.edu/~koopman/stack_computers/). *Ece.cmu.edu*. Retrieved 2017-10-08.
28. *Introduction to A Series Systems* (http://www.bitsavers.org/pdf/burroughs/A-Series/MCP_3.6/1170057_Introduction_to_A_Series_Systems_3.6_Apr86.pdf) (PDF). Burroughs Corporation. April 1986. Retrieved 2022-07-07.
29. "Design and Implementation of an Efficient Stack Machine" (http://www.jopdesign.com/doc/stack.pdf) (PDF). *Jopdesign.com*. Retrieved 2017-10-08.
30. Chatterji, Satrajit; Ravindran, Kaushik. "BOOST: Berkeley's Out of Order Stack Thingie" (https://www.researchgate.net/publication/228556746). *Research Gate*. Kaushik Ravindran. Retrieved 2016-02-16.
31. Bergh, Arndt; Keilman, Keith; Magenheimer, Daniel; Miller, James (December 1987). "HP3000 Emulation on HP Precision Architecture Computers" (http://www.hpl.hp.com/hpjournal/pdfs/IssuePDFs/1987-12.pdf) (PDF). *Hewlett-Packard Journal*. Hewlett Packard: 87–89. Retrieved 2017-10-08.
32. Migrating a CISC Computer Family onto RISC via Object Code Translation. Kristy Andrews, Duane Sand: Proceedings of ASPLOS-V, October 1992
33. 8051 CPU Manual, Intel, 1980
34. Shi, Yunhe; Gregg, David; Beatty, Andrew; Ertle, M. Anton. "Virtual Machine Showdown: Stack vs. Register Machine" (http://usenix.org/events/vee05/full_papers/p153-yunhe.pdf) (PDF). *Usenix.org*. Retrieved 2017-10-08.
35. Davis, Brian; Beatty, Andrew; Casey, Kevin; Gregg, David; Waldron, John. "The Case for Virtual Register Machines" (http://www.scss.tcd.ie/David.Gregg/papers/Gregg-SoCP-2005.pdf) (PDF). *Scss.tcd.ie*. Retrieved 2017-10-08.
36. Bornstein, Dan (2008-05-29). "Presentation of Dalvik VM Internals" (https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0) (PDF). p. 22. Retrieved 2010-08-16.
37. "The Implementation of Lua 5.0" (http://www.lua.org/doc/jucs05.pdf) (PDF). *Lua.org*. Retrieved 2017-10-08.
38. "The Virtual Machine of Lua 5.0" (http://www.inf.puc-rio.br/~roberto/talks/lua-ll3.pdf) (PDF). *Inf.puc-rio.br*. Retrieved 2017-10-08.
39. "Branch Prediction and the Performance of Interpreters - Don't Trust Folklore" (https://hal.inria.fr/hal-01100647/document). *Hal.inria.fr*. Retrieved 2017-10-08.

External links

- Homebrew CPU in an FPGA (http://www.excamera.com/sphinx/fpga-j1.html) — homebrew stack machine using FPGA
- Mark 1 FORTH Computer (http://www.holmea.demon.co.uk/Mk1/Architecture.htm) — homebrew stack machine using discrete logical circuits
- Mark 2 FORTH Computer (http://www.holmea.demon.co.uk/Mk2/Architecture.htm) — homebrew stack machine using bitslice/PLD
- Second-Generation Stack Computer Architecture (http://fpgacpu.ca/stack/Second-Generation_Stack_Computer_Architecture.pdf) — Thesis about the history and design of stack machines.
