J1: a small Forth CPU Core for FPGAs

James Bowman
Willow Garage
Menlo Park, CA
jamesb@willowgarage.com

***Abstract*— This paper describes a 16-bit Forth CPU core, intended for FPGAs. The instruction set closely matches the Forth programming language, simplifying cross-compilation. Because it has higher throughput than comparable CPU cores, it can stream uncompressed video over Ethernet using a simple software loop. The entire system (source Verilog, cross compiler, and TCP/IP networking code) is published under the BSD license. The core is less than 200 lines of Verilog, and operates reliably at 80 MHz in a Xilinx Spartan®-3E FPGA, delivering approximately 100 ANS Forth MIPS.**

I. INTRODUCTION

The J1 is a small CPU core for use in FPGAs. It is a 16-bit von Neumann architecture with three basic instruction formats. The instruction set of the J1 maps very closely to ANS Forth. The J1 does not have:

- condition registers or a carry flag
- pipelined instruction execution
- 8-bit memory operations
- interrupts or exceptions
- relative branches
- multiply or divide support.

Despite these limitations it has good performance and code density, and reliably runs a complex program.

II. RELATED WORK

While there have been many CPUs for Forth, three current designs stand out as options for embedded FPGA cores:

MicroCore [1] is a popular configurable processor core targeted at FPGAs. It is a dual-stack Harvard architecture, encodes instructions in 8 bits, and executes one instruction in two system clock cycles. A call requires two of these instructions: a push literal followed by a branch to Top-of-Stack (TOS). A 32-bit implementation with all options enabled runs at 25 MHz - 12.5 MIPS - in a Xilinx Spartan-2S FPGA.

b16-small [2], [3] is a 16-bit RISC processor. In addition to dual stacks, it has an address register A, and a carry flag C. Instructions are 5 bits each, and are packed 1-3 in each word. Byte memory access is supported. Instructions execute at a rate of one per cycle, except memory accesses and literals which take one extra cycle. The b16 assembly language resembles Chuck Moore's ColorForth. FPGA implementations of b16 run at 30 MHz.

eP32 [4] is a 32-bit RISC processor with deep return and data stacks. It has an address register (X) and status register (T). Instructions are encoded in six bits, hence each 32-bit word contains five instructions. Implemented in TSMC's 0.18μm CMOS standard library the CPU runs at 100 MHz, providing 100 MIPS if all instructions are short. However a jump or call instruction causes a stall as the target instruction is fetched, so these instructions operate at 20 MIPS.

III. THE J1 CPU

A. Architecture

This description follows the convention that the top of stack is T, the second item on the stack is N, and the top of the return stack is R.

J1's internal state consists of:

- a 33 deep $\times$ 16-bit data stack
- a 32 deep $\times$ 16-bit return stack
- a 13-bit program counter

There is no other internal state: the CPU has no condition flags, modes or extra registers.

Memory is 16-bits wide and addressed in bytes. Only aligned 16-bit memory accesses are supported: byte memory access is implemented in software. Addresses 0-16383 are RAM, used for code and data. Locations 16384-32767 are used for memory-mapped I/O.

The 16-bit instruction format (table I) uses an unencoded hardwired layout, as seen in the Novix NC4016 [5]. Like many other stack machines, there are five categories of instructions: literal, jump, conditional jump, call, and ALU.

Literals are 15-bit, zero-extended to 16-bit, and hence use a single instruction when the number is in the range 0-32767. To handle numbers in the range 32768-65535, the compiler follows the immediate instruction with `invert`. Hence the majority of immediate loads take one instruction.

All target addresses - for call, jump and conditional branch - are 13-bit. This limits code size to 8K words, or 16K bytes. The advantages are twofold. Firstly, instruction decode is simpler because all three kinds of instructions have the same format. Secondly, because there are no relative branches, the cross compiler avoids the problem of range overflow in `resolve`.

Conditional branches are often a source of complexity in CPUs and their associated compiler. J1 has a single instruction that tests and pops T, and if $T = 0$ replaces the current PC with the 13-bit target value. This instruction is the same as `0branch` word found in many Forth implementations, and is of course sufficient to implement the full set of control structures.

ALU instruction have multiple fields:

field	width	action
T'	4	ALU op, replaces T, see table II
$T \rightarrow N$	1	copy T to N
$R \rightarrow PC$	1	copy R to the PC
$T \rightarrow R$	1	copy T to R
dstack $\pm$	2	signed increment data stack
rstack $\pm$	2	signed increment return stack
$N \rightarrow [T]$	1	RAM write

Table III shows how these fields may be used together to implement several Forth primitive words. Hence each of these words map to a single cycle instruction. In fact J1 executes all of the frequent Forth words - as measured by [6] and [7] - in a single clock cycle.

As in the Novix and SC32 [8] architectures, consecutive ALU instructions that use different functional units can be merged into a single instruction. In the J1 this is done by the assembler. Most importantly, the `;` instruction can be merged with a preceding ALU operation. This trivial optimization, together with the rewriting of the last call in a word as a jump, means that the `;` (or `exit`) instruction is free in almost all cases, and reduces our measured code size by about 7%, which is in line with the static instruction frequency analysis in [7].

The CPU's architecture encourages highly-factored code:

- the call instruction is always single-cycle
- `;` and `exit` are usually free
- the return stack is 32 elements deep

B. Hardware Implementation

Execution speed is a primary goal of the J1, so particular attention needs to be paid to the critical timing path. This is the path from RAM read, via instruction fetch to the computation of the new value of T. Because the ALU operations (table II) do not depend on any fields in the instruction, the computation of these values can be done in parallel with instruction fetch and decode, figure 1.

The data stack D and return stack R are implemented as small register files; they are not resident in RAM. This conserves RAM bandwidth, allowing `@` and `!` to operate in a single cycle. However, this complicates implementation of `pick` and `roll`.

Our FPGA vendor's embedded SRAM is dual-ported. The core issues an instruction read every cycle (port a) and a memory read from T almost every cycle (port b), using the latter only in the event of an `@` instruction. In case of a memory write, however, port b does the memory write in the following cycle. Because of this, `@` and `!` are single cycle operations[1].

In its current application - an embedded Ethernet camera - the core interfaces with an Aptina imager and an open source Ethernet MAC using memory mapped I/O registers. These registers appear as memory locations in the $4000-$7FFF range so that their addresses can be loaded in a single literal instruction.

[1]the assembler inserts a `drop` after `!` to remove the second stack parameter

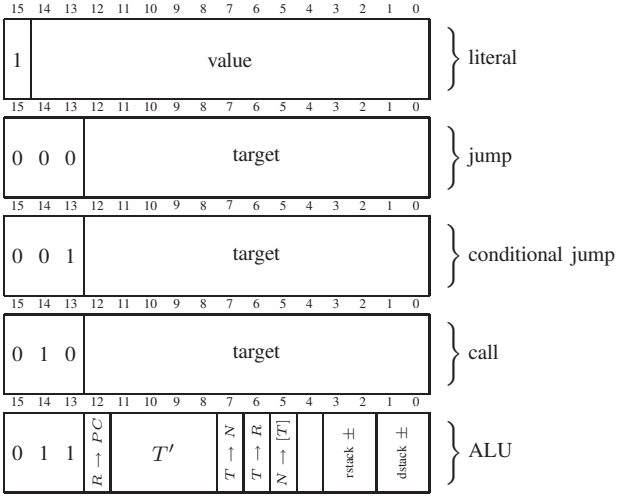

TABLE I: Instruction encoding

code	operation
0	T
1	N
2	$T + N$
3	$T$and$N$
4	Tor$N$
5	Txor$N$
6	$\sim T$
7	$N = T$
8	$N < T$
9	Nrshift$T$
10	$T - 1$
11	R
12	$[T]$
13	Nlshift$T$
14	depth
15	Nu<T

TABLE II: ALU operation codes

word	T'	$T \rightarrow N$	$R \rightarrow PC$	$T \rightarrow R$	dstack $\pm$	rstack $\pm$	$N \rightarrow [T]$
dup	T	•			+1	0	
over	N	•			+1	0	
invert	$\sim T$				0	0	
+	$T + N$				-1	0	
swap	N	•			0	0	
nip	T				-1	0	
drop	N				-1	0	
;	T		•		0	-1	
>r	N			•	-1	+1	
r>	R	•		•	+1	-1	
r@	R	•		•	+1	0	
@	$[T]$				0	0	
!	N				-1	0	•

TABLE III: Encoding of some Forth words.

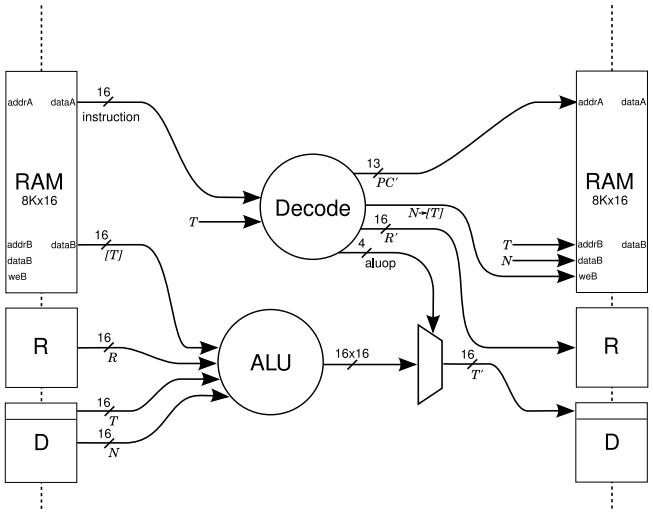

Fig. 1: The flow of a single instruction execution. ALU operation proceeds in parallel with instruction fetch and decode. Bus widths are in bits.

C. System Software

Because the target machine matches Forth so closely. the cross assembler and compiler are relatively simple. These tools run under gforth [9]. The compiler generates native code, sometimes described as subroutine-threaded with inline code expansion [8].

Almost all of the core words are written in pure Forth, the exceptions are `pick` and `roll`, which must use assembly code because the stack is not accessible in regular memory. Much of the core is based on eforth [10].

D. Application Software

The J1 is part of a system which reads video from an Aptina image sensor and sends it as UDP packets over Ethernet. The PR2 robot running ROS [11] uses six of these cameras, two in stereo pairs in the head and one in each arm.

The main program implements a network stack (MAC interface, Ethernet, IP, ARP, UDP, TCP, DHCP, DNS, HTTP, NTP, TFTP and our own UDP-based camera control protocol), handles I^2C, SPI, and RS-232 interfaces, and streams video data from the image sensor.

The heart of the system is this inner loop, which moves 32 bits of data from the imager to the MAC:

```
begin
    begin MAC_tx_ready @ until
    pixel_data @ MAC_tx_0 !
    pixel_data @ MAC_tx_1 !
    1- dup 0=
until
```

IV. Results

The J1 performs well in its intended application. This section attempts to quantify the improvements in code density and system performance.

Static analysis of our application gives the following instruction breakdown:

instruction	usage
conditional jump	4%
jump	8%
literal	22%
call	29%
ALU	35%

An earlier version of the system used a popular RISC soft-core [12] based on the Xilinx MicroBlaze®architecture, and was written in C. Hence it is possible to compare code sizes for some representative components. Also included are some tentative results from building the same Forth source on MicroCore.

component	MicroBlaze	J1	MicroCore
	code size (bytes)		
I^2C	948	132	113
SPI	180	104	105
flash	948	316	370
ARP responder	500	122	–
entire program	16380	6349	–

The J1 code takes about 62% less space than the equivalent MicroBlaze code. Since the code store allocated to the CPU is limited to 16 Kbytes, the extra space freed up by switching to the J1 has allowed us to add features to the camera program. As can be seen, J1's code density is similar to that of the MicroCore, which uses 8-bit instructions.

While J1 is not a general purpose CPU, and its only performance-critical code section is the video copy loop shown above, it performs quite well, delivering about 3X the system performance of the previous C-based system running on a MicroBlaze-compatible CPU.

V. Conclusion

By using a simple Forth CPU we have made a more capable, better performing and more robust product.

Some directions for our future work: increasing the clock rate of the J1; using J1 in other robot peripherals; implementing the ROS messaging system on the network stack.

Our source code and documentation are available at: `http://www.ros.org/wiki/wge100_camera_firmware`

VI. Acknowledgments

I would like to thank Blaise Glassend for the original implementation of the camera hardware.

References

[1] K. Schleisiek, "MicroCore," in *EuroForth*, 2001.
[2] B. Paysan. `http://www.jwdt.com/~paysan/b16.html`.
[3] B. Paysan, "b16-small – Less is More," in *EuroForth*, 2004.

[4] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in *Canadian Conference on Electrical and Computer Engineering*, pp. 518–521, 2007.
[5] E. Jennings, “The Novix NC4000 Project,” *Computer Language*, vol. 2, no. 10, pp. 37–46, 1985.
[6] D. Gregg, M. A. Ertl, and J. Waldron, “The Common Case in Forth Programs,” in *EuroForth*, 2001.
[7] P. J. Koopman, Jr., *Stack computers: the new wave*. New York, NY, USA: Halsted Press, 1989.
[8] J. Hayes, “SC32: A 32-Bit Forth Engine,” *Forth Dimensions*, vol. 11, no. 6, p. 10.
[9] A. Ertl, B. Paysan, J. Wilke, and N. Crook. `http://www.jwdt.com/~paysan/gforth.html`.
[10] B. Muench. `http://www.baymoon.com/~bimu/forth/`.
[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “Ros: an open-source robot operating system,” in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, (Kobe, Japan), May 2009.
[12] S. Tan. `http://www.aeste.my/aemb`.