EXTENDING POWER ISA FOR SEARCH WITH SVP64*

Luke Kenneth Casson Leighton†,
Toshaan Bharvani, Vantosh,
Konstantinos Margaritis, VectorCamp

Abstract

Under normal circumstances Search and AI algorithm implementers are left with the unenviable task of optimising code for hardware that they had no input into its design, and if by chance the original designers of the hardware or crucially the ISA happened to have tested a particular algorithm and thought hard about it, software writers might end up with optimal performance and power consumption. If however they step outside of that box there is nothing that they can do other than to search for alternative hardware on which to optimally implement a Search algorithm, or to tolerate the sub-par performance and power usage. Whilst SVP64 will ultimately likely suffer this same fate at some point, the opportunity exists during this early phase its lifecycle to look closely at Search and AI algorithms to see if there is anything that can be done. Early exploration showed that a paralleliseable Vector strncpy can be implemented in as little as ten SVP64 Assembler instructions.

INTRODUCTION TO SVP64

The basic principle of SVP64 is to turn Vectorisation into a type of Scalar Loop Construct. This is what SIMD and normal Vector ISAs look like:

```
for i in range(SIMDlength):
    VR(RT)[i] = VR(RA)[i] + VR(RB)[i]
```

This is what SVP64 looks like:

```
for i in range(VL):
   GPR(RT+i) = GPR(RA+i) + GPR(RB+i)
```

Note immediately that as a direct consequence of defining **VL** directly in terms of *Elements* instead of the Vector Register bit-width, the known-issue of always having to have a Vector Loop at all times in order to guarantee Binary-executable Portability, SVP64 programmers may write simple loops - including SIMD ones, freed from Power-of-Two limitations - in just two lines of code[1].

```
for i in range(VL):
    if predicate.bit[i] clear: # skip?
       continue
    GPR(RT+i) = GPR(RA+i) + GPR(RB+i)
    if CCTest(GPR(RT+i)) fails: # end?
        VL = i # truncate the Vector
        break
```

On top of these very basic but already-profound[2] beginnings, Predication and Conditional-Exit can be added. Predication is found in every GPU ISA, and Conditional-Exit is a 50-year invention dating back to Zilog Z80 CPIR and LDIR.

Additionally the concept may be introduced from ARM SVE and RISC-V RVV "Fault-First" on Load and Store, where if an Exception would occur then the Hardware informs the programmer that the Vector operation is truncated:

```
for i in range(VL):
   if predicate.bit[i] clear:
       continue
   EffectiveAddress = GPR(RA+i) + Immediate
     if Exception@(EffectiveAddress):
       if i == 0: RAISE Exception
       else: VL = i; break # truncate
   GPR(RT+i) = Mem@(EffectiveAddress)
```

The important facet of both these "Conditional Truncation" constructs is that there exists a Contract between Programmer and Hardware. The Programmer requests *up to* a certain Vector Length, and the Hardware informs the Programmer exactly how much work was *actually* carried out. The most important aspect is the hardware *informing* the Programmer how far it got, in an implicit but 100% Deterministic fashion, by truncating the Vector Length.

With these two complementary and inter-related constructs, all the usual hassle with SIMD - often compensated for with hard-coded dedicated "Memory copy" or "String copy" instructions that cannot be leveraged for any other purpose, goes away.

STRNCPY

strncpy [2] presents some unique challenges for an ISA and hardware, the primary being that in a SIMD (parallel) context, strncpy operates in bytes where SIMD operates in power-of-two multiples only. PackedSIMD is the worst offender: PredicatedSIMD is marginally better[3]. If SIMD Load and Store has to start on an Aligned Memory location, which is a common limitation, things get even worse. The operations that were supposed to speed up algorithms have to have "preamble" and "postamble" to take care of the corner-cases.

Worse, a naive SIMD ISA cannot have Conditional inter-relationships. In well-defined ISAs, 128-byte or greater

* NGI Search, EU Grant 101069364

† lkcl@lkcl.net

[1] with the proviso that the Programmer must be mindful of both the starting point and what they set MAXVL to. Hardware will helpfully remind them of any Register File overruns by happily throwing an Illegal Instruction

[2] caveats: with hardware and ISA Architectural requirements that deal with the increased Dependency Hazard Management, too detailed to list in full in this document, the most important being that the total number of registers be a fixed **and mandatory** Standards-defined quantity

[3] caveat: if designed properly, as was done successfully in ARM SVE

LOADs either succeed in full or they fail in full. If the strncpy subroutine happens to copy from the last few bytes in memory, SIMD LOADs are the worst thing to use. We need a way to Conditionally terminate the LOAD and inform the Programmer, and this is where (as in ARM SVE) Load-Fault-First comes into play.

However even this is not enough: once LOADed it is necessary to first spot the NUL character, and once identified to then begin copying NUL characters from that point onwards.

```
for (i = 0; i < n && src[i] != '\0'; i++)
    dest[i] = src[i];
for ( ; i < n; i++)
   dest[i] = '\0';
```

Leaving aside the prior issue that LOADing beyond the point where the NUL was should not even have been attempted in the first place, performing such a conditional NUL-character search in a SIMD ISA is typically extremely convoluted. A usual approach would be to perform a Parallel compare against NUL (easy enough) followed by an instruction that then searches sequentially for the first fail, followed by another instruction that explicitly truncates the Vector Length, followed finally by the actual STORE.

```
mtspr 9, 3   # move r3 to CTR
addi 0,0,0   # initialise r0 to zero
# chr-copy loop starts here:
# for (i=0; i<n && src[i] != '\0'; i++)
#    dest[i] = src[i];
# VL (and r1) = MIN(CTR,MAXVL=4)
setvl 1,0,MAXVL,0,1,1
# load VL bytes (update r10 addr)
sv.lbzu/pi *16, 1(10)
# compare against zero, truncate VL
sv.cmpi/ff=eq/vli *0,1,*16,0
# store VL bytes (update r12 addr)
sv.stbu/pi *16, 1(12)
# test CTR, stop if cmpi failed
sv.bc/all 0, *2, -0x1c
# zeroing loop starts here:
#   for ( ; i < n; i++)
#       dest[i] = '\0';
# VL (and r1) = MIN(CTR,MAXVL=4)
setvl 1,0,MAXVL,0,1,1
# store VL zeros (update r12 addr)
sv.stbu/pi 0, 1(12)",
# decrement CTR by VL, stop at zero
sv.bc 16, *0, -0xc
```

All of the sequential-search-and-truncate is part of the Data-Dependent Fail-First Mode that is a first-order construct in SVP64. When applied to the **sv.cmpi** instruction, which produces a Vector of Condition Codes as opposed to just one for the Scalar **cmpi** instruction), the search for the NUL character truncates the Vector Length at the required point, such that the next instruction (STORE) is already set up to copy up to and including the NUL (if one was indeed found).

The next most important addition to SVP64 is a Vector-aware Branch-Conditional instruction. Where **sv.cmpi** had created a Vector of Condition Codes, **sv.bc/all** will only Branch back to continue loading/copying of bytes iff no NUL was found and there are more characters to copy.

A normal ISA would not have such parallel Condition Code Branch instructions. It would perhaps have a way to reduce a batch of parallel Condition Codes down to a *single* Condition Code, and then use a *Scalar* Branch-Conditional. Additionally the opportunity is taken to reduce the **CTR** Special Purpose Register by the (run-time truncated) Vector Length, saving the Programmer from having to explicitly copy the Vector Length into a GPR, explicitly subtract that from a copy of CTR, then explicitly copy the subtraction result back into CTR.

The end-result of these enhancements is an overwhelmingly-compact *general-purpose* Vector ISA that effectively did nothing more complex than bring back 50-year-old concepts from 8-bit micro-processors. With the high reduction in program size comes a high ”bang-per-buck” ratio that often allows the core inner loop (in this case the entire strncpy subroutine) to fit into a single L1 Cache Line, avoiding TLB misses and thus significantly saving on power consumption as well as potential operational delays.

CONCLUSION

Our goal as part of NGI Search is to validate that the approach taken above works across multiple algorithms. VectorScan [1] was chosen as a high-value library due to the sheer overwhelming complexity needed for other ISAs. libc6 was also chosen as it is such a low-level library that any Search algorithm utilising it would benefit from increased compactness and efficiency.

SVP64 chose very deliberately a design paradigm that only general-purpose constructs be added. There are no hard-coded dedicated specialist ”memory copy” instructions, with the crucial side-effect that a **strncpyW** instruction (a UCS-2 variant of strncpy) is simply a matter of using general-purpose 16-bit cmp and 16-bit LOAD/STORE instead of general-purpose 8-bit cmp and 8-bit LOAD/STORE.

Thus it is anticipated that future programmers may be freed from many of the limitations inherent in other ISAs, by being able to express high-level language constructs much more directly, cleanly and clearly in SVP64 Assembler. All whilst retaining an all-important *general-purpose* Sequential Programming paradigm.

REFERENCES

[1] VectorScan, `https://github.com/VectorCamp/vectorscan`

[2] `https://git.libre-soc.org/?p=openpower-isa.git;a=blob;f=src/openpower/decoder/isa/test_caller_svp64_ldst.py;hb=HEAD`