arm

Introduction to Arm SVE and Gem5 simulator

Javier Setoain

Arm Research

Software and Large Scale Systems Group

Outline

About Arm SVE

- Introduction
- Basic SVE architecture
- Vector Length Agnostic Vectorization

About gem5

- Introduction
- Simulation models
- Operating System modes
- Using gem5
- Opportunities and limitations of simulator

Introduction

Basic vectorization

```
void example01( int *restrict a, const int *b,
                                 const int *c, long N)
{
        long i;
        for (i = 0; i < N; ++i)
                a[i]   = b[i]   + c[i];
}
```

Introduction

Basic vectorization

```
void example01( int *restrict a, const int *b,
                                 const int *c, long N)
{
        long i;
        for (i = 0; i < N; i+=4) {
                a[i]   = b[i]   + c[i];
                a[i+1] = b[i+1] + c[i+1];
                a[i+2] = b[i+2] + c[i+2];
                a[i+3] = b[i+3] + c[i+3];
    }
}
```

Introduction

Basic vectorization

```
void example01( int *restrict a, const int *b,
                                 const int *c, long N)
{
        long i;
        for (i = 0; i < N - 3; i+=4) {
                a[i]   = b[i]   + c[i];
                a[i+1] = b[i+1] + c[i+1];
                a[i+2] = b[i+2] + c[i+2];
                a[i+3] = b[i+3] + c[i+3];
    }
        // loop tail
        for (; i < N; ++i)
                a[i]   = b[i]   + c[i];
}
```

Introduction

Basic vectorization

```
void example01_neon(int *restrict a, const
                    int *b, const int *c, long N)
{
        long i;
        for (i = 0; i < N - 3; i += 4) {
                int32x4_t vb = vld1q_s32(b + i);
                int32x4_t vc = vld1q_s32(c + i);
                int32x4_t va = vaddq_s32(vb, vc);
                vst1q_s32(a + i, va);
        }
        // loop tail
        for (; i < N; ++i)
                a[i]   = b[i]   + c[i];
}
```

Introduction

Basic vectorization

```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x8 is the loop induction variable 'i'
  mov     x8, xzr
  subs    x9, x3, 3             # x9 = N -3
  b.ls    .loop_tail_preheader  # jump to loop tail if N <= 3
.vector_body:
  ldr     q0, [x1, x8, lsl 4]   # load 4 elements from 'b+i'
  ldr     q1, [x2, x8, lsl 4]   # load 4 elements from 'c+i'
  add     v0.4s, v1.4s, v0.4s   # add the vector
  str     q0, [x0, x8, lsl 4]   # store 4 elements in 'a+i'
  add     x8, x8, 4             # increment 'i' by 4
  cmp     x8, x9                # compare i with N - 3
  b.lo    .vector_body          # keep looping if i < N-3
.loop_tail_preheader:
  cmp     x8, x3                # compare the loop counter with N
  b.hs    .function_exit        # if greater or equal N, terminate
.loop_tail:
  ldr     w12, [x1, x8, lsl 2]
  ldr     w13, [x2, x8, lsl 2]
  add     w12, w13, w12
  str     w12, [x0, x8, lsl 2]
  cmp     x8, x3
  b.lo    .loop_tail            # keep looping until no elements remain
.function_exit:
  ret
```


arm

Introduction

NEON

- NEON works with 128-bit vectors
  - Way too small for today's HPC standards
- It doesn't support conditional execution
  - Limited vectorization possibilities
- It can operate only on contiguous memory positions
- Should we create NEON+?
  - And when that's not enough, NEON++?
  - And then NEON+++?
  - …

Introduction

Scalable Vector Extension: Vector Length Agnostic Programming

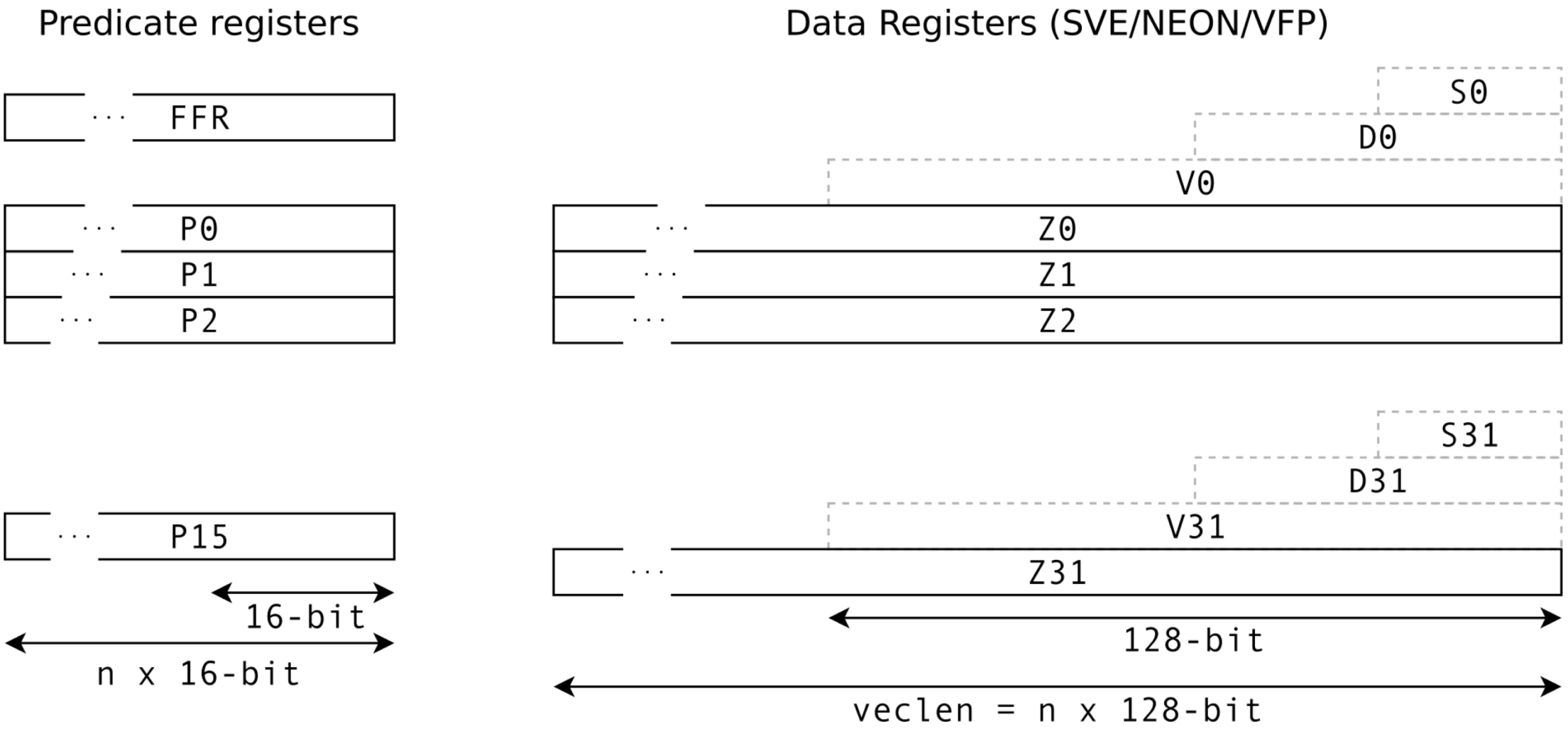
Problems:

- Different partners have different vectorization needs
- ISA namespace is limited
  - Ever growing SIMD extensions are not practical
- It’s costly to vectorize your code for every available SIMD extension
- Traditional SIMD instruction sets are media-processing focused

Solution:

- Scalable Vector Extensions: a partner can license the vector length that best suites their needs
  - 128 – 2048 bits (in 128-bit increments)
- Vector Length Agnostic (VLA) programming
  - The same vector code will run regardless of vector size
- HPC focus (also server workloads in general, e.g.: ML)

Basic SVE architecture



arm

Basic SVE architecture

- Data types
  - Integer
    - 8-bit, 16-bit, 32-bit, 64-bit
  - Floating point
    - half-precision*, single precision, double precision
- Memory operations
  - Contiguous load/store
  - Structure load/store (AoS to SoA and back)
  - Non-contiguous load/store
    - Stride, Gather and Scatter
  - Non-temporal accesses
    - Improved stream operations
  - Speculative accesses
  - Load & Replicate
  - ...

Basic SVE architecture

- Arithmetic and logic operations
  - All your classics
  - Reductions
  - Some exotic instructions, e.g.: population count.
- Predicate-specific instructions
  - Generation
  - Manipulation
  - Partition
  - …
- Permute operations
  - Extract elements
  - Compact
  - Permute
  - …

Vector Length Agnostic Vectorization

NEON vectorization

```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x8 is the loop induction variable 'i'
  mov     x8, xzr
  subs    x9, x3, 3             # x9 = N -3
  b.ls    .loop_tail_preheader  # jump to loop tail if N <= 3
.vector_body:
  ldr     q0, [x1, x8, lsl 4]   # load 4 elements from 'b+i'
  ldr     q1, [x2, x8, lsl 4]   # load 4 elements from 'c+i'
  add     v0.4s, v1.4s, v0.4s   # add the vector
  str     q0, [x0, x8, lsl 4]   # store 4 elements in 'a+i'
  add     x8, x8, 4             # increment 'i' by 4
  cmp     x8, x9                # compare i with N - 3
  b.lo    .vector_body          # keep looping if i < N-3
.loop_tail_preheader:
  cmp     x8, x3                # compare the loop counter with N
  b.hs    .function_exit        # if greater or equal N, terminate
.loop_tail:
  ldr     w12, [x1, x8, lsl 2]
  ldr     w13, [x2, x8, lsl 2]
  add     w12, w13, w12
  str     w12, [x0, x8, lsl 2]
  cmp     x8, x3
  b.lo    .loop_tail            # keep looping until no elements remain
.function_exit:
  ret
```


arm

Vector Length Agnostic Vectorization

SVE vectorization

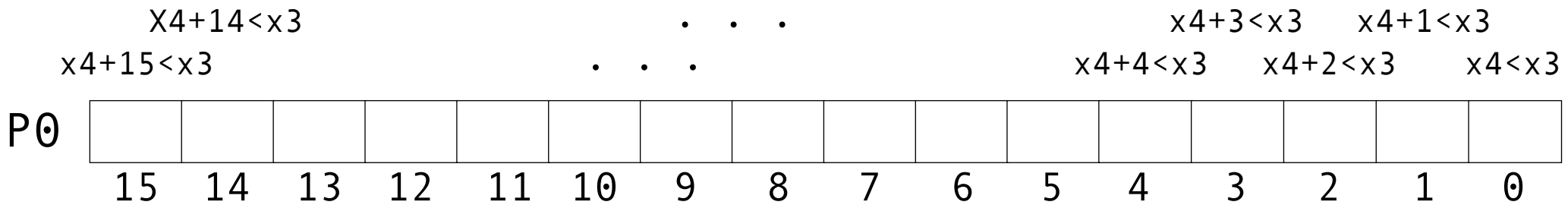
```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
  mov     x4, 0 # set 'i=0'
  b       cond  # branch to 'cond'
loop_body:
  ld1w    z0.s, p0/z, [x1, x4, lsl 2]# load vector z0 from address 'b + i'
  ld1w    z1.s, p0/z, [x2, x4, lsl 2]# same, but from 'c + i' into vector z1
  add     z0.s, p0/m, z0.s, z1.s     # add the vectors
  st1w    z0.s, p0, [x0, x4, lsl 2]  # store vector z0 at 'a + I'
  incw    x4                         # increment 'i' by number of words in a vector
cond:
  whilelt p0.s, x4, x3 # build the loop predicate p0, as p0.s[idx] = (x4+idx) < x3
                       # it also sets the condition flags
  b.first loop_body    # branch to 'loop_body' if the first bit in the predicate
                       # register 'p0' is set
  ret
```


arm

Vector Length Agnostic Vectorization

SVE vectorization

```
whilelt p0.s, x4, x3
```

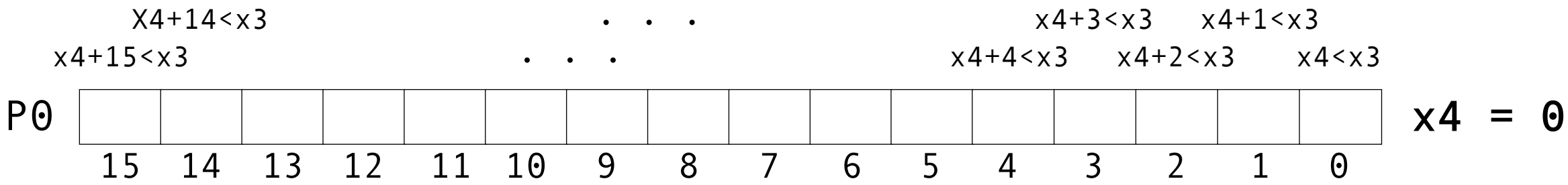


arm

Vector Length Agnostic Vectorization

SVE vectorization

```
whilelt p0.s, x4, x3 #[x3 = 20]
```

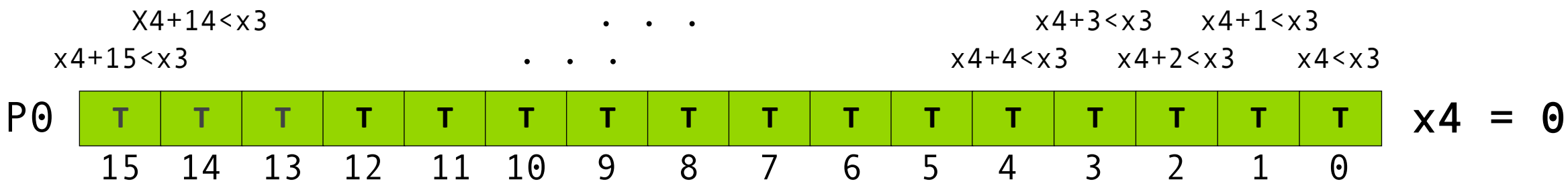


arm

Vector Length Agnostic Vectorization

SVE vectorization

```
whilelt p0.s, x4, x3 #[x3 = 20]
```



arm

Vector Length Agnostic Vectorization

SVE vectorization

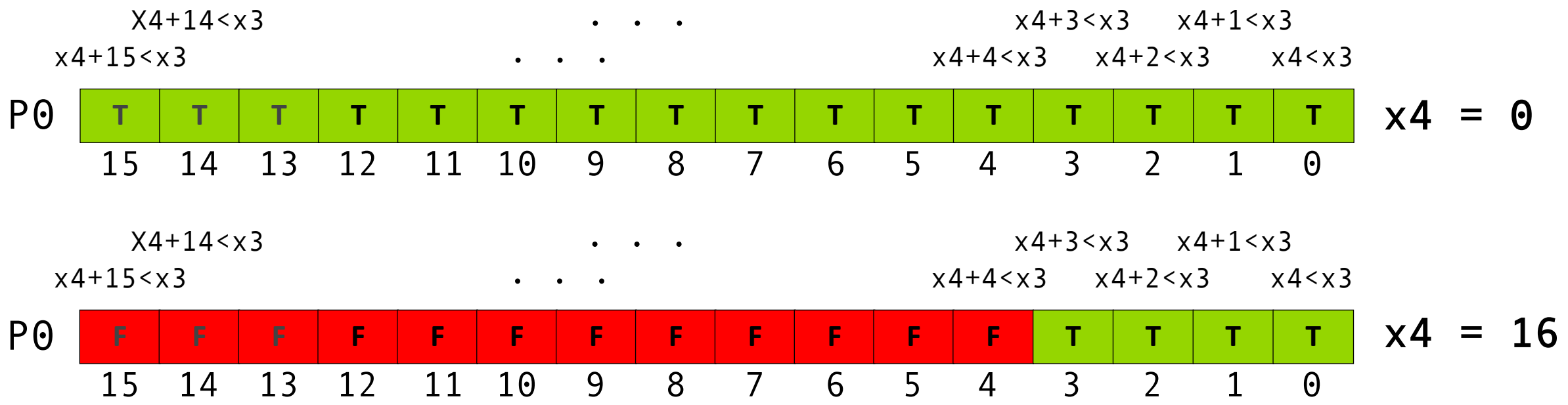
```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
  mov     x4, 0 # set 'i=0'
  b       cond  # branch to 'cond'
loop_body:
  ld1w    z0.s, p0/z, [x1, x4, lsl 2]# load vector z0 from address 'b + i'
  ld1w    z1.s, p0/z, [x2, x4, lsl 2]# same, but from 'c + i' into vector z1
  add     z0.s, p0/m, z0.s, z1.s     # add the vectors
  st1w    z0.s, p0, [x0, x4, lsl 2]  # store vector z0 at 'a + I'
  incw    x4                         # increment 'i' by number of words in a vector
cond:
  whilelt p0.s, x4, x3 # build the loop predicate p0, as p0.s[idx] = (x4+idx) < x3
                       # it also sets the condition flags
  b.first loop_body    # branch to 'loop_body' if the first bit in the predicate
                       # register 'p0' is set
  ret
```


arm

Vector Length Agnostic Vectorization

SVE vectorization

```
whilelt p0.s, x4, x3 #[x3 = 20]
```



arm

Vector Length Agnostic Vectorization

SVE vectorization

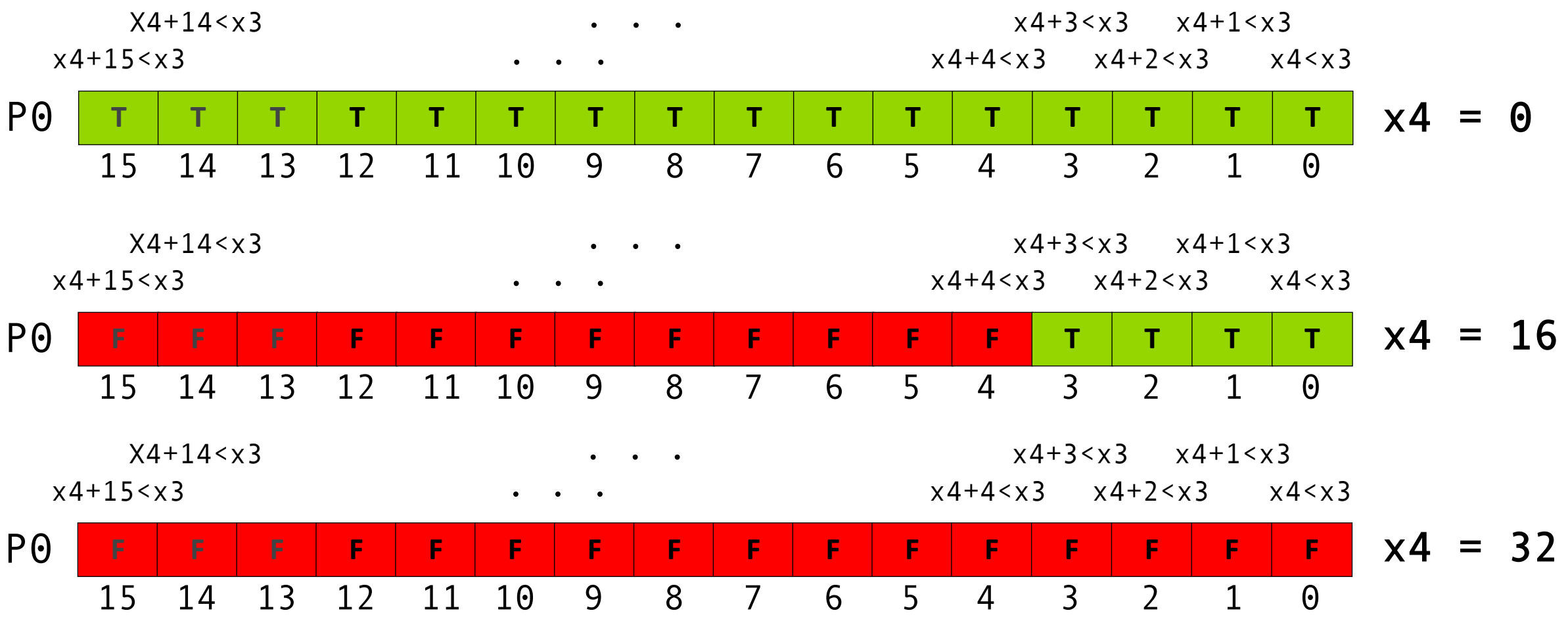
```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
  mov     x4, 0 # set 'i=0'
  b       cond  # branch to 'cond'
loop_body:
  ld1w    z0.s, p0/z, [x1, x4, lsl 2]# load vector z0 from address 'b + i'
  ld1w    z1.s, p0/z, [x2, x4, lsl 2]# same, but from 'c + i' into vector z1
  add     z0.s, p0/m, z0.s, z1.s     # add the vectors
  st1w    z0.s, p0, [x0, x4, lsl 2]  # store vector z0 at 'a + I'
  incw    x4                         # increment 'i' by number of words in a vector
cond:
  whilelt p0.s, x4, x3 # build the loop predicate p0, as p0.s[idx] = (x4+idx) < x3
                       # it also sets the condition flags
  b.first loop_body    # branch to 'loop_body' if the first bit in the predicate
                       # register 'p0' is set
  ret
```


arm

Vector Length Agnostic Vectorization

SVE vectorization

```
whilelt p0.s, x4, x3 #[x3 = 20]
```



arm

Vector Length Agnostic Vectorization

SVE vectorization

```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'i'
  mov     x4, 0 # set 'i=0'
  b       cond  # branch to 'cond'
loop_body:
  ld1w    z0.s, p0/z, [x1, x4, lsl 2]# load vector z0 from address 'b + i'
  ld1w    z1.s, p0/z, [x2, x4, lsl 2]# same, but from 'c + i' into vector z1
  add     z0.s, p0/m, z0.s, z1.s     # add the vectors
  st1w    z0.s, p0, [x0, x4, lsl 2]  # store vector z0 at 'a + I'
  incw    x4                         # increment 'i' by number of words in a vector
cond:
  whilelt p0.s, x4, x3 # build the loop predicate p0, as p0.s[idx] = (x4+idx) < x3
                       # it also sets the condition flags
  b.first loop_body    # branch to 'loop_body' if the first bit in the predicate
                       # register 'p0' is set
  ret
```


arm

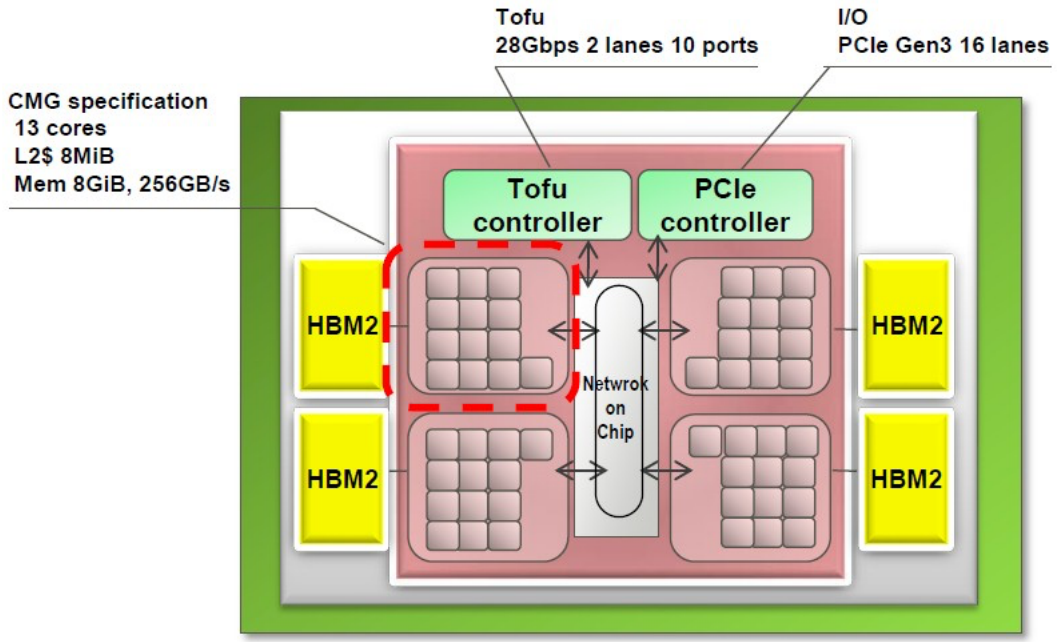
SVE Programming

Documentation and tools

- ISA specification: The Scalable Vector Extension for Armv8-A
- Explained examples: A sneak peek into SVE and VLA programming
- Intrinsics
  - Documentation: Arm C Language Extensions for SVE
  - Examples: Arm Scalable Vector Extensions and application to Machine Learning
- Compilers & auto-vectorization
  - GCC (-ftree-vectorize or -O3 for basic block vectorizer)
    - -ftree-vectorize-verbose=2 will give information about vectorization success/failure
  - Arm Compiler for HPC (at least -O2)
    - #pragma directives can be used to give the vectorizer hints
      - #pragma omp parallel for simd
      - #pragma clang loop vectorize(enable) …
  - Cray
  - Fujitsu

SVE implementations: Fujitsu's A64FX

- Architecture features
  - Armv8.2-A (AArch64 only)
  - SVE 512-bit
  - 4 x 12 compute cores + 4 "assistant" cores
  - 4 x 8 GiB HBM2
  - Fujitsu's Tofu interconnect
    - 6D Mesh/Torus
    - 28 Gbps x 2 lanes x 10 ports
  - 16 lanes of PCIe 3.0
- 7nm FinFET
  - 8.786 billion transistors
  - 594 package signal pins
- Peak Performance
  - > 2.7 TFLOPs (> 90% @ DGEMM)
  - Memory B/W 1024 GB/s (> 80% @ Stream Triad)



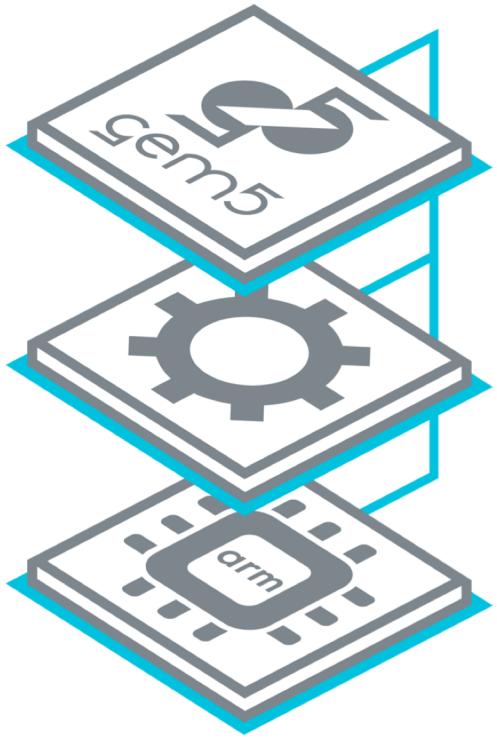
arm


arm
gem5 simulation

Introduction

What is gem5?

- gem5 is a simulation infrastructure
  - You can use it to build simulators
- gem5 is a research tool
  - Simulators are a cost-effective way to evaluate ideas
- gem5 models various system components
  - Different CPUs, interconnects, memory subsystems, …
- gem5 also models the interactions between these components



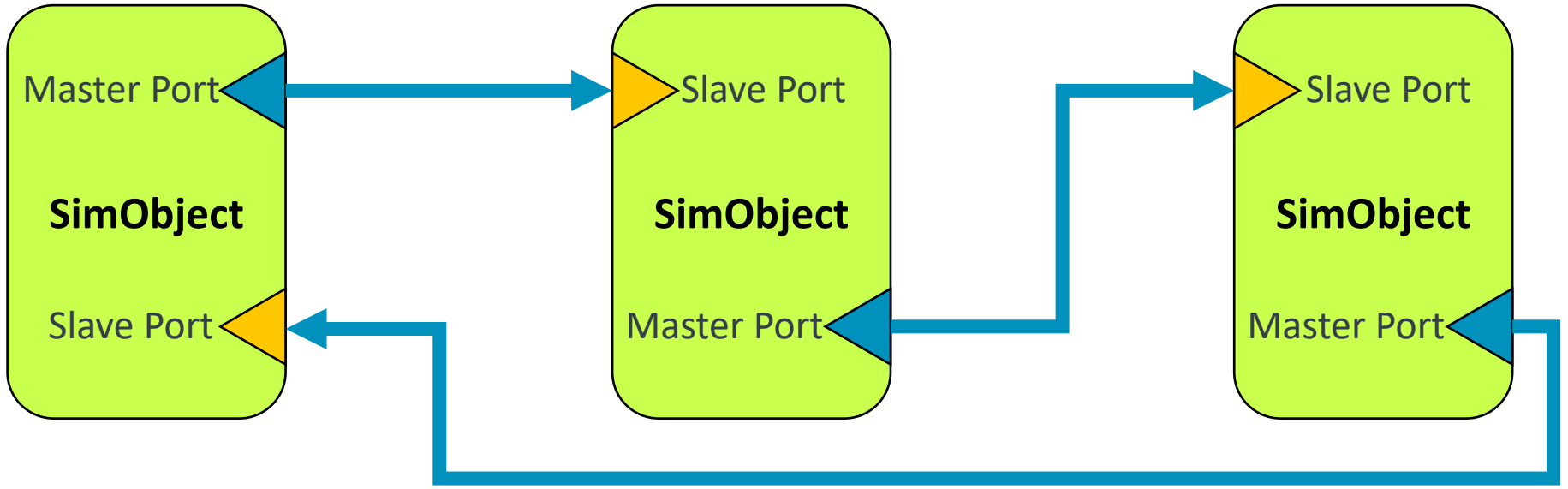
Arm Research Enablement Kit is a guide through Arm-based system modeling using the gem5 simulator and a 64-bit processor model based on Armv8-A.

https://developer.arm.com/research/research-enablement/system-modeling

Introduction

Simulation model

- Three basic components:
  - SimObject: the simulated component
  - Master Port: outgoing endpoint through which the SimObject issues requests
  - Slave Port: incoming endpoint through which the SimObject receives requests



arm

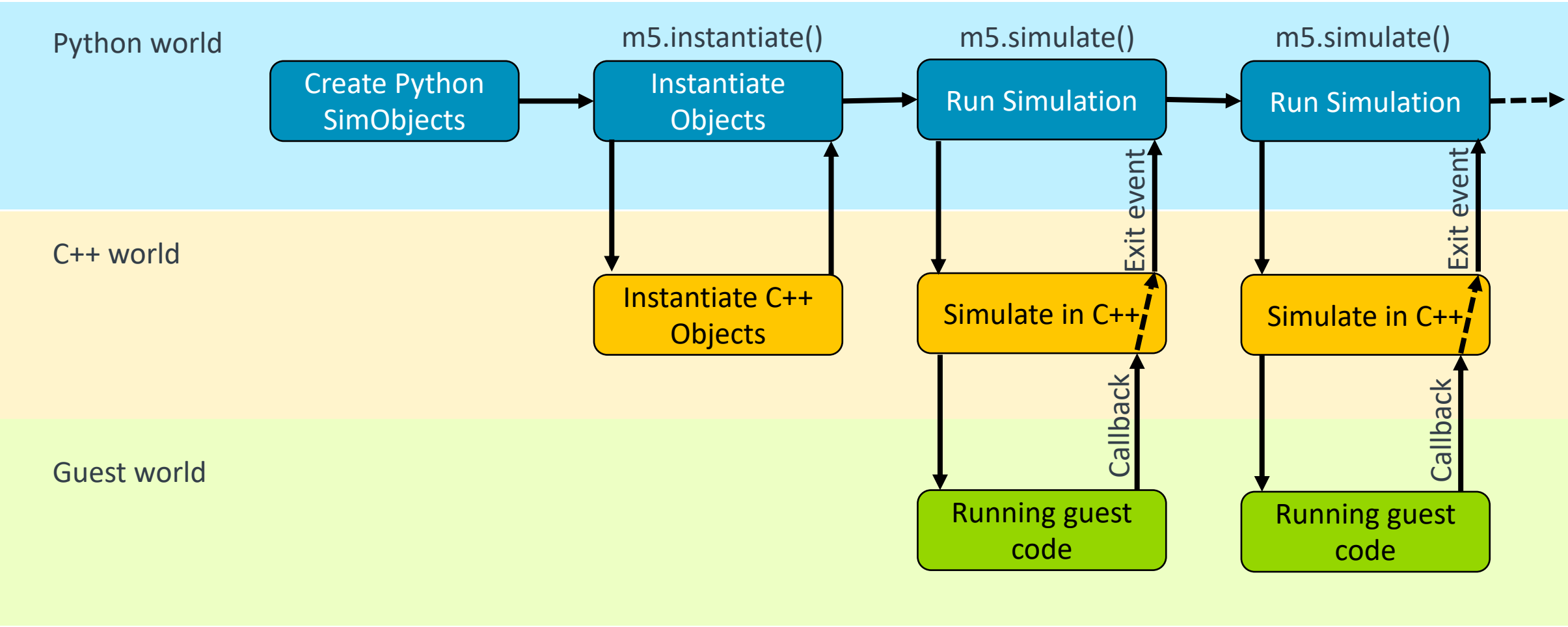
Introduction

Simulation setup

- Gem5 is conceptually a Python library implemented in C++
  - Configured by instantiating Python classes with matching C++ classes
    - From Python, instantiate SimObject objects and arrange them by interconnecting their ports
  - Parameters exposed as attributes in Python
  - Running is controlled from Python but implemented in C++
- Two steps: Configuration and execution
  - Configuration ends with a call to instantiate the C++ world
  - Parameters cannot be changed after the C++ world has been created
  - Running as a standard discrete event timing model (logical time measured in ‘ticks’)

Introduction

Simulation flow

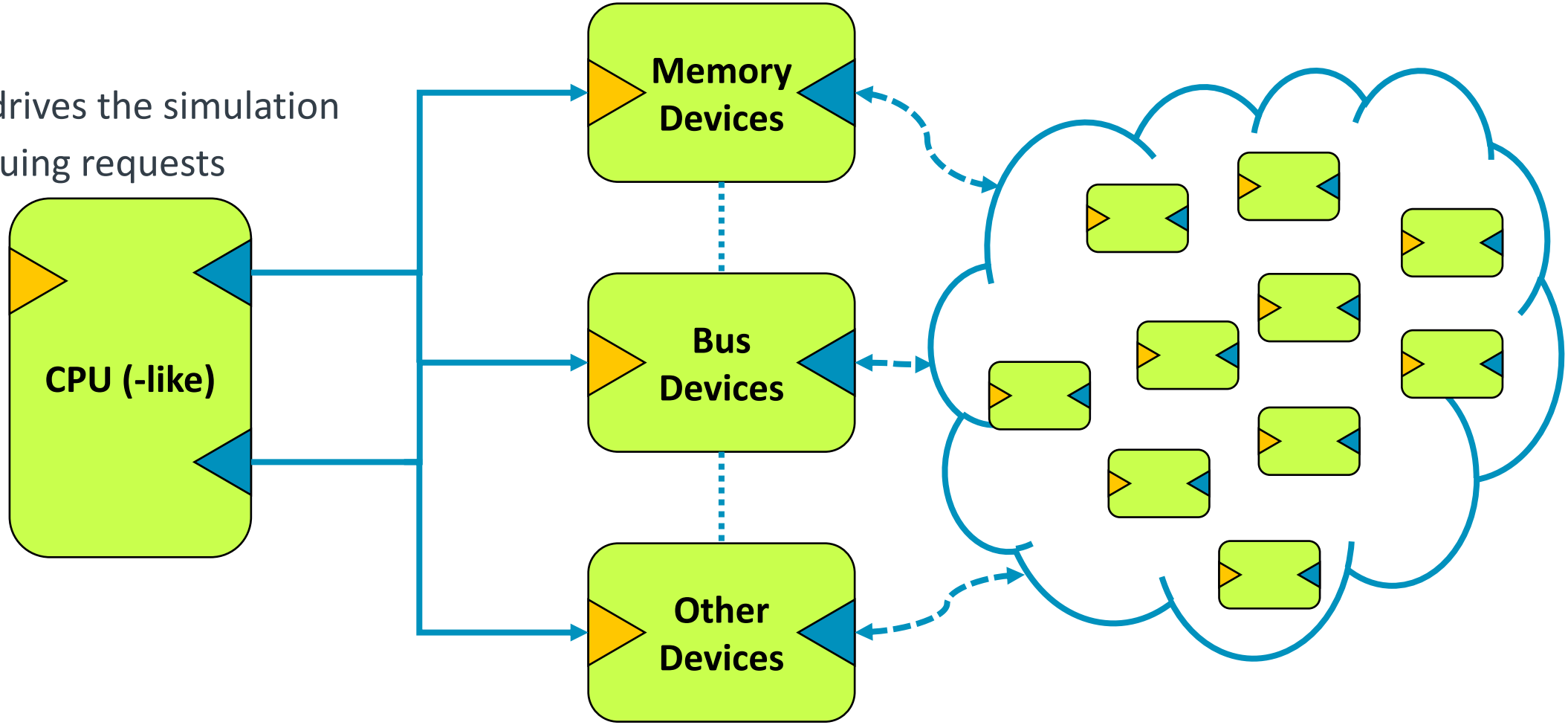


arm

Simulation modes

General system overview

CPU drives the simulation by issuing requests



arm

Simulation modes

Timing model

- Timing
  - Accurate timing information
  - Queueing delay and resource contention
  - Requests are processed asynchronously
  - Slow
- Atomic
  - Very limited timing information
  - Request are processed instantaneously
  - Useful for fast-forwarding and warp-up
  - Fastest
- Functional
  - Limited timing information
  - Requests are processed instantaneously
  - Can co-exist with Timing or Atomic elements
  - Used to model I/O and connect debug interfaces

Operating system modes

Question: How do we deal with the operating system?

- System calls
- Libraries
- Process scheduling
- Inter-process communication
- …

Operating system modes

Two options

- Simulated OS
- Emulated OS

Operating system modes

Full-system Simulation

- Full-system
  - Boot an operating system
  - Models bare hardware, including devices -> simulated inputs (keyboard, VNC) and output (UART, frame buffer)
  - Interrupts, exceptions, fault handlers
  - Privileged instructions
  - Bigger (and slower) model

Operating system modes

Syscall Emulation

- Syscall Emulation
  - Run individual applications or set of applications on MP
  - Models user-level ISA plus common system calls
  - System calls emulated, typically by calling the host OS
  - Simplified address translation model
  - No process scheduling
  - Binaries have to be statically compiled (no dynamic link support)
  - Smaller, more streamlined (faster model) model
  - Misses timing for all OS interactions

Using gem5

Prerequisites

Operating system:

- OSX, Linux
- Limited support for Windows 10 with Linux environment

Software dependencies:

- git
- python 2.7 (dev. Packages)
- gcc 4.8 or clang 3.1 (or newer)
- swig 2.0.4 or newer
- zlib
- m4
- make

Optional dependencies:

- dtc (to compile device trees for full system)
- Armv8 cross compiler (to compile workloads)
- protobuf 2.1 or later (trace capture and playback)
- python-pydot (to generate system diagrams)

Using gem5

Downloading

Clone ***development*** repository

```
~$ git clone https://gem5.googlesource.com/public/gem5
```

Update the cloned repository:

```
~$ git pull
```

Using gem5

Compiling

```
~$ scons build/ARM/gem5.opt -j4
```

- Guest architecture
- Several architectures in the source tree
- Most common ones are:
  - **ARM**
  - **NULL**
    - For trace-driven simulation
  - **X86**
    - Popular in academia but very strange timing behavior

- Optimization level:
  - **debug** : Debug symbols, no/few optimizations
  - **opt** : Debug symbols + most optimizations
  - **fast** : No symbols + even more optimizations

Using gem5

Running syscall emulation

```
~$ build/ARM/gem5.opt configs/example/se.py --num-cpus=4 --caches --l2cache --mem-type=DDR4_2400_16x4
--mem-size=2GB --cpu-type=DerivO3CPU -c <binary> -o "<cmd. line options>"
```

Runs <*binary*> on a generic system with an out-of-order CPU and 2GB of DDR4

- DerivO3CPU is a *timing* model, so it will run in *timing* mode
- Timing CPUs require at least L1 caches (it will complain otherwise)
- se.py is only a sample configuration, you can write your own configuration scripts
  - Don't look into it to see how to do that, it's unnecessarily complex
  - `configs/learning_gem5/part1` is a much better starting point

Using gem5

Statistics

- Output @ `m5out` (unless otherwise specified)

Sample stats.txt:

```
---------- Begin Simulation Statistics ----------
final_tick                                18397500 # Number of ticks from beginning of simulation
(restored from checkpoints and never reset)
host_inst_rate                               9479 # Simulator instruction rate (inst/s)
host_mem_usage                            2331956 # Number of bytes of host memory used
host_op_rate                                10997 # Simulator op (including micro ops) rate (op/s)
host_seconds                                 0.53 # Real time elapsed on the host
host_tick_rate                            34773999 # Simulator tick rate (ticks/s)
sim_freq                               1000000000000 # Frequency of simulated ticks
sim_insts                                    5014 # Number of instructions simulated
sim_ops                                      5818 # Number of ops (including micro ops) simulated
sim_seconds                                0.000018 # Number of seconds simulated
sim_ticks                                 18397500 # Number of ticks simulated
…
system.mem_ctrls.bw_read::total   1381057209 # Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_inst_read::.cpu0.inst 960130453 # Instruction read bandwidth from this
memory (bytes/s)
```


arm

Opportunities and limitations of simulators

Good practices

- Think about what you want to evaluate and make sure everything that can have a meaningful impact has been adequately modeled
  - E.g.: if you want to have an idea of your B/W consumption, make sure everything between your CPU and your memory is properly arranged
  - The better your base model, the more you can trust it
- Relative information is more valuable
  - E.g.: better than raw B/W numbers, relative improvements when you make a change are likely to give you more valuable information
- Testing the limits of your algorithms and/or architectures can provide valuable information
  - E.g.: configure an infinite B/W memory, how much faster can I run? That's going to be (roughly) your limit for how much better you can perform as memory improves
- Use vendor-provided simulators when available
  - Riken offers access to their A64FX simulator (NDA)

Pitfalls

- gem5 doesn't simulate an x86 or Arm processor, gem5 simulates a gem5 processor that runs that ISA
  - Avoid direct performance extrapolations
    - It can be useful as a reference, but don't take it as an absolute prediction
- If you're going to draw conclusions from a specific subsystem, make sure that said subsystem is modeled with enough detail
  - E.g.: memory subsystem and interactions with surrounding devices (this includes the CPU)
- If numbers look too good/odd/bad to be true, they probably aren't
  - Making sure things make sense is important to draw accurate conclusions

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

Extra SVE examples

- Vectorize with control dependencies

Vectorisation: control dependencies

```
void example02(int *restrict a, const int *b,
               const int *c, long N, const int *d)
{
  long i;
  for (i = 0; i < N; ++i)
    if (d[i] > 0)
      a[i] = b[i] + c[i];
}
```

Vectorisation: control dependencies

```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'd', x5 is 'i'
  mov     x5, 0 # set 'i = 0'
  b       cond
loop_body:
  ld1w    z4.s, p0/z, [x4, x5, lsl 2] # load a vector from 'd + i'
  cmpgt   p1.s, p0/z, z4.s, 0         # compare greater than zero
                                      # p1.s[idx] = z4.s[idx] > 0
  # from now on all the instructions depending on the 'if' statement are
  # predicated with 'p1'
  ld1w    z0.s, p1/z, [x1, x5, lsl 2]
  ld1w    z1.s, p1/z, [x2, x5, lsl 2]
  add     z0.s, p1/m, z0.s, z1.s
  st1w    z0.s, p1, [x0, x5, lsl 2]
  incw    x5
cond:
  whilelt p0.s, x5, x3
  b.first loop_body
  ret
```

Extra SVE examples

- Vectorize reduction & conditional reduction with zeroing and merging

Vectorisation: reduction

```
int example02(int *a, int *b, long N)
{
  long i;
  int s = 0;
  for (i = 0; i < N; ++i)
    s += a[i];
  return s;
}
```

Conditional vector reduction

```
int example02(int *a, int *b, long N)
{
  long i;
  int s = 0;
  for (i = 0; i < N; ++i)
    if (b[i])
      s += a[i];
  return s;
}
```

Conditional vector reduction: zeroing & merging

```
  mov     x5, 0   # set 'i = 0’
  mov     z0.s, 0 # set the accumulator 's' to zero
  b       cond
loop_body:
  ld1w    z4.s, p0/z, [x1, x5, lsl 2] # load a  vector
                                      # at 'b + i'
  cmpne   p1.s, p0/z, z4.s, 0         # compare non zero
                                      # into predicate 'p1'
  # from now on all the instructions depending on the 'if' statement are
  # predicated with 'p1'
  ld1w    z1.s, p1/z, [x0, x5, lsl 2]
  add     z0.s, p1/m, z0.s, z1.s      # the inactive lanes
                                      # retain the partial sums
                                      # of the previous iterations

  incw    x5
cond:
  whilelt p0.s, x5, x3
  b.first loop_body
  ptrue p0.s
  saddv d0, p0, z0.s # signed add words across the lanes of z0, and place the
                     # scalar result in d0
  mov w0, v0.s[0]
  ret
```

Extra SVE examples

- Vectorize reduction

Non-contiguous memory access

```
void example03(int *restrict a, const int *b,
               const int *c, long N, const int *d)
{
   long i;
   for (i = 0; i < N; ++i)
      a[i] = b[d[i]] + c[i];
}
```

Non-contiguous memory access: gather & scatter

```
# x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'd', x5 is 'i'
  mov     x5, 0
  b       cond
loop:
  ld1w    z1.s, p0/z, [x4, x5, lsl 2]
  ld1w    z0.s, p0/z, [x1, z1.s, sxtw 2] # load a vector
                                         # from  'x1 + sxtw(z1.s) << 2'
  ld1w    z1.s, p0/z, [x2, x5, lsl 2]
  add     z0.s, p0/m, z0.s, z1.s
  st1w    z0.s, p0, [x0, x5, lsl 2]
  incw    x5
cond:
  whilelt p0.s, x5, x3
  b.first loop
  ret
```


arm

Extra SVE examples

- Vectorize string operations with speculative memory accesses

NULL-terminated string operations?

```
void strcpy(char *restrict dst, const char *src)
{
  while (1) {
    *dst = *src;
    if (*src == '\0') break;
    src++; dst++;
  }
}
```

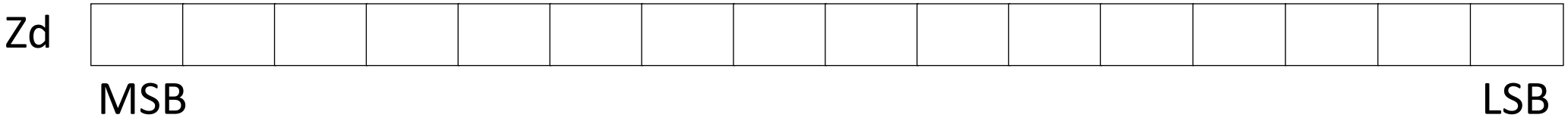
NULL-terminated string operations?

const char* src = “Hello, world”;

arm

NULL-terminated string operations?

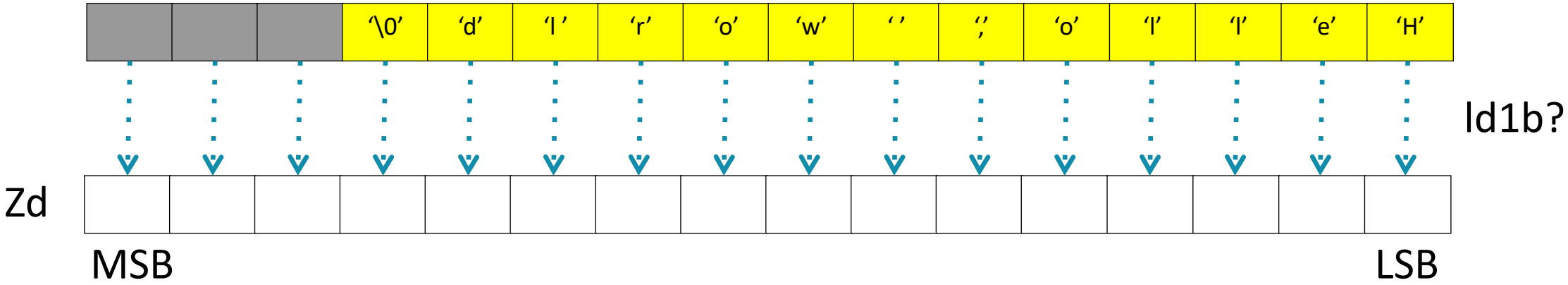
const char* src = “Hello, world”;



arm

NULL-terminated string operations?

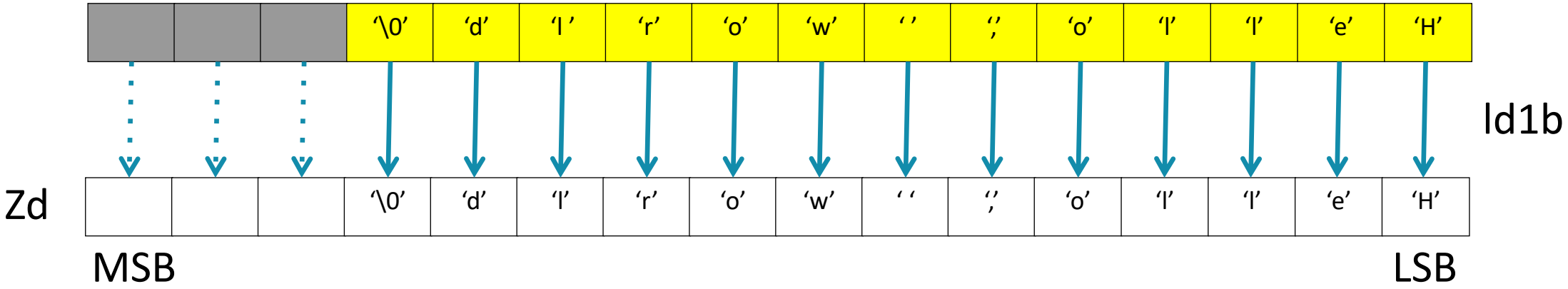
const char* src = “Hello, world”;



arm

NULL-terminated string operations?

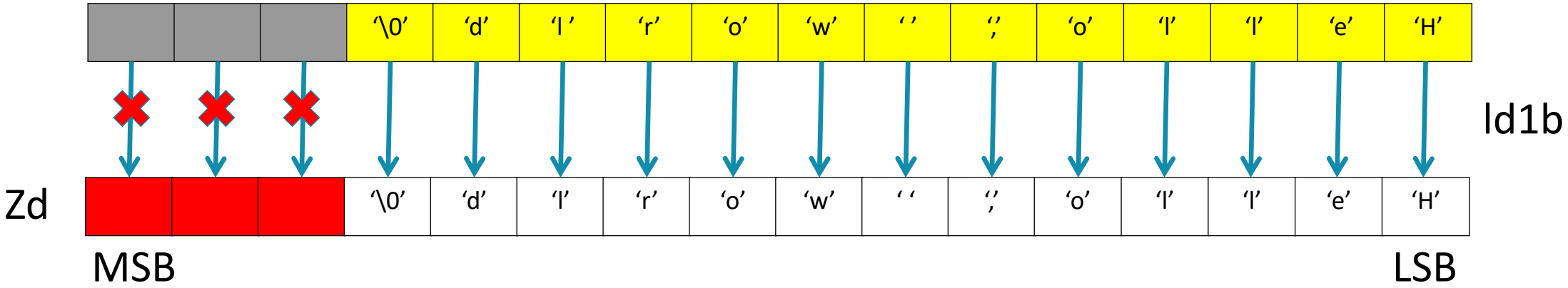
```
const char* src = "Hello, world";
```



arm

NULL-terminated string operations?

```
const char* src = "Hello, world";
```



- Segmentation fault

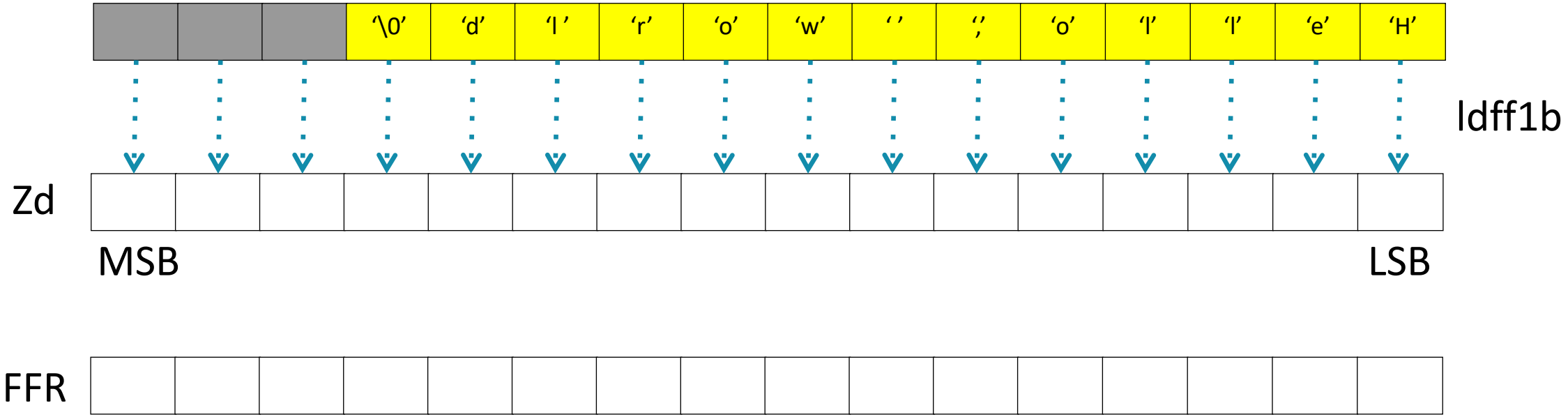
arm

Dealing with unknown loop bounds: Speculative Load

- First Fault Register (FFR)
  - Like predicate registers: 1 bit per byte in a vector register
  - Special instruction to load the whole vector speculatively: ldff1
  - Keep track of which accesses fault
  - If all accesses fault, it’s a real fault and throws an exception

Dealing with unknown loop bounds: Speculative Load

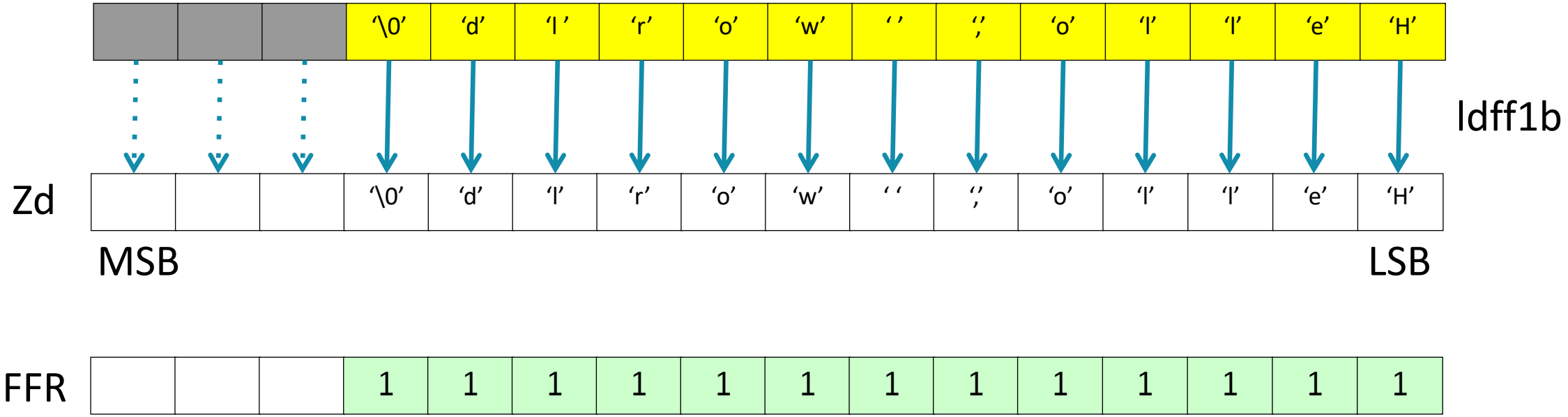
const char* src = “Hello, world”;



arm

Dealing with unknown loop bounds: Speculative Load

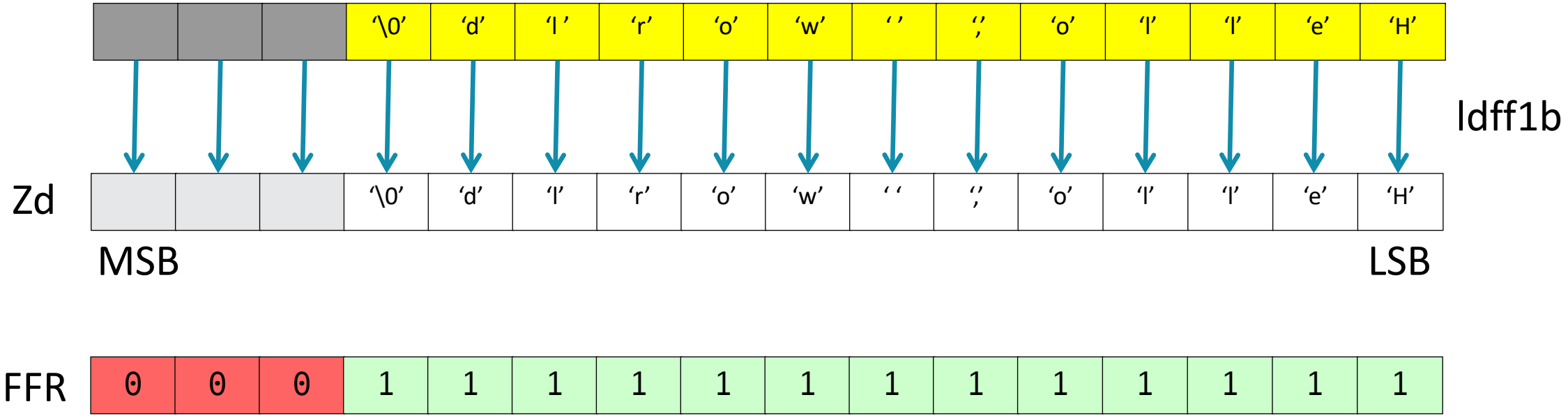
```
const char* src = "Hello, world";
```



arm

Dealing with unknown loop bounds: Speculative Load

const char* src = “Hello, world”;



arm

Dealing with unknown loop bounds: Speculative Load

```
void strcpy(char *restrict dst, const char *src)
{
  while (1) {
    *dst = *src;
    if (*src == '\0') break;
    src++; dst++;
  }
}
```

Dealing with unknown loop bounds: Speculative Load

```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```

Dealing with unknown loop bounds: Speculative Load

```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```

Set all bits in FFR to 1

arm

Dealing with unknown loop bounds: Speculative Load

```
sve_strcpy:
  mov      x2, 0
  ptrue    p2.b
loop:
  setffr
  ldff1b   z0.b, p2/z, [x1, x2]
  rdffr    p0.b, p2/z
  cmpeq    p1.b, p0/z, z0.b, 0
  brka     p0.b, p0/z, p1.b
  st1b     z0.b, p0, [x0, x2]
  incp     x2, p0.b
  b.none   loop
  ret
```

Speculative load (fill FFR)

Dealing with unknown loop bounds: Speculative Load

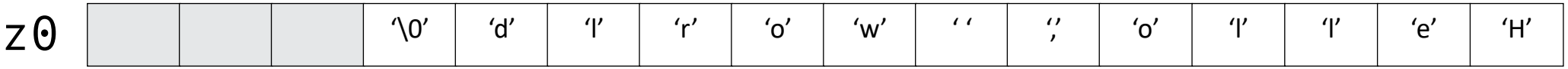
```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```

Read FFR into a predicate register

Dealing with unknown loop bounds: Speculative Load

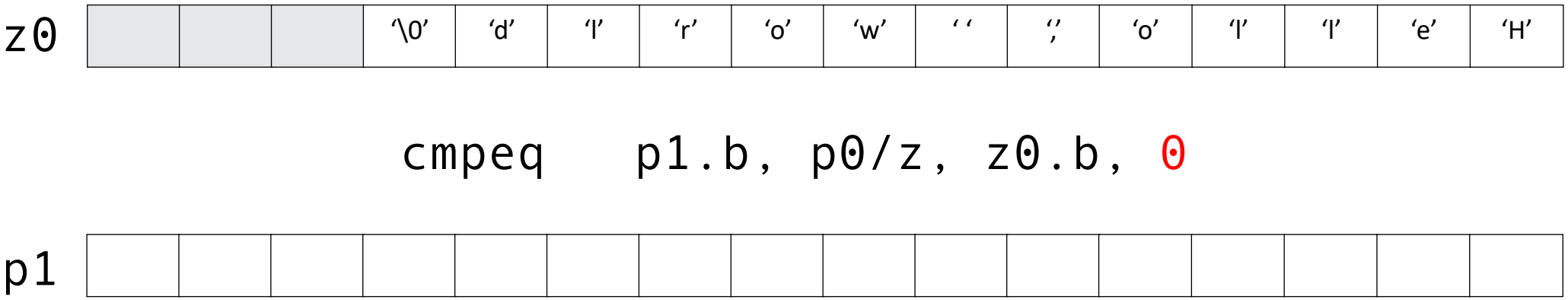
```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```

Dealing with unknown loop bounds: Speculative Load



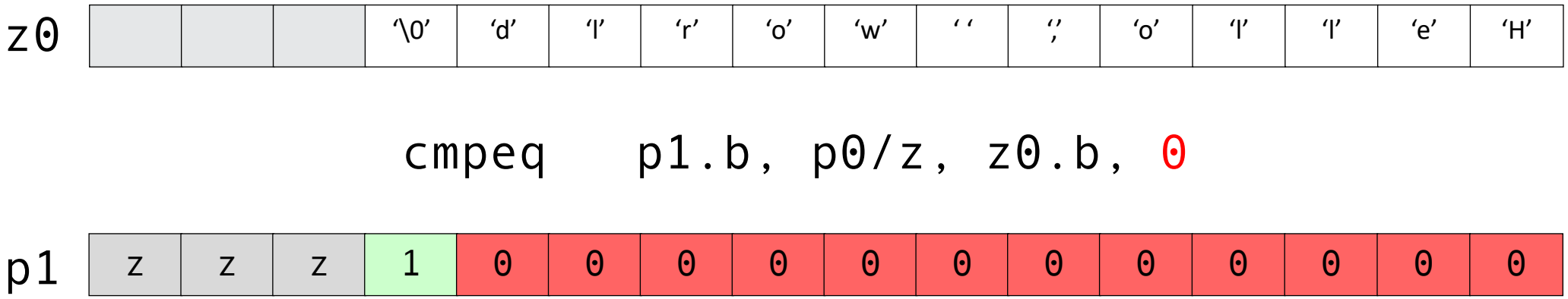
arm

Dealing with unknown loop bounds: Speculative Load



arm

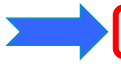
Dealing with unknown loop bounds: Speculative Load



arm

Dealing with unknown loop bounds: Speculative Load

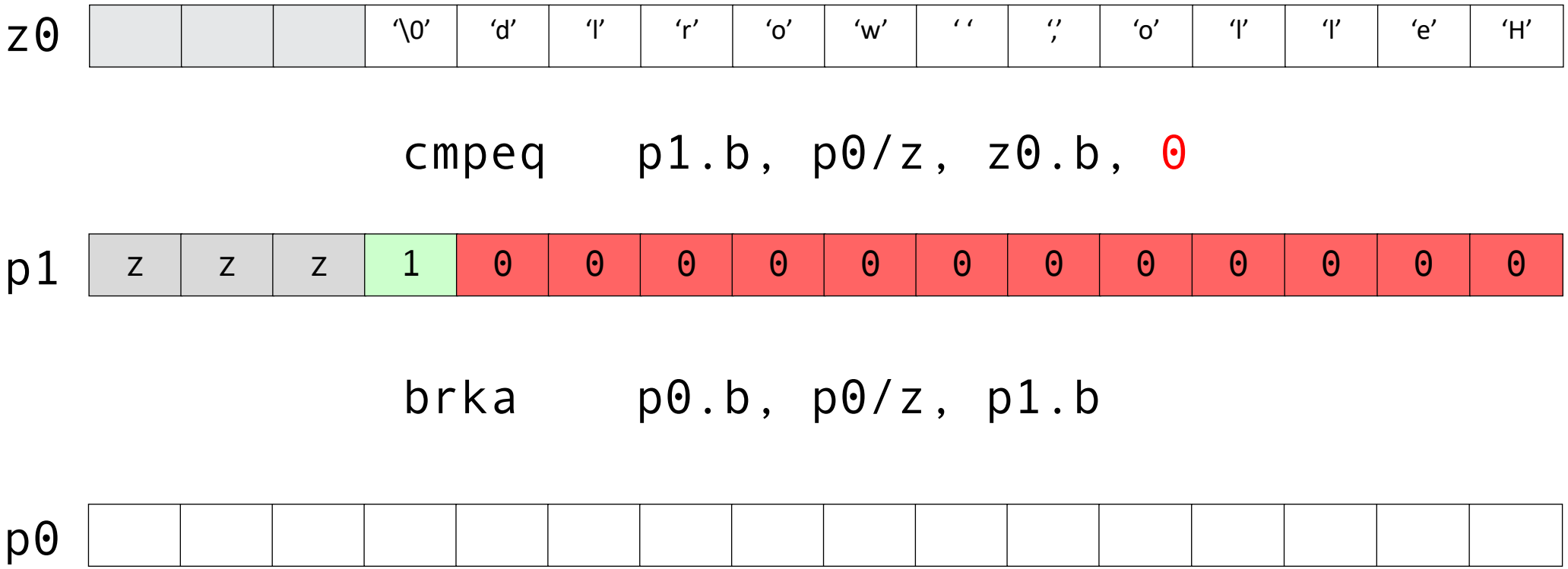
```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```



Read FFR into a predicate register

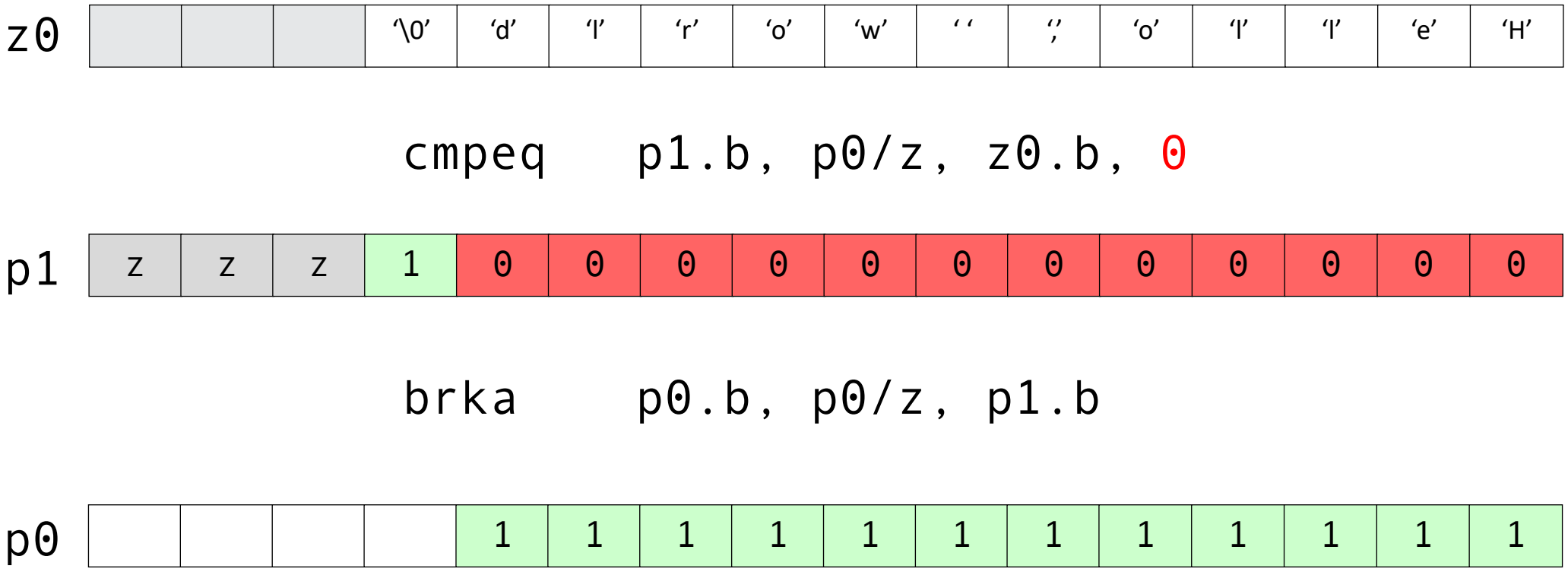
arm

Dealing with unknown loop bounds: Speculative Load



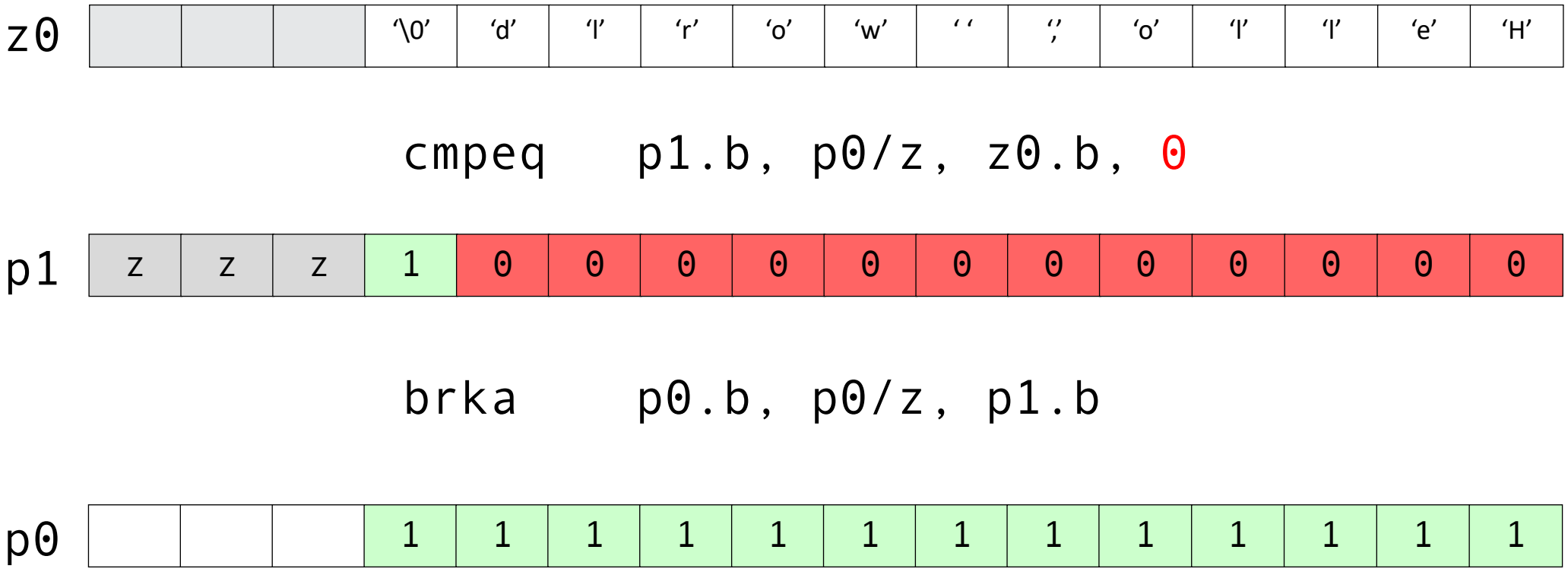
arm

Dealing with unknown loop bounds: Speculative Load



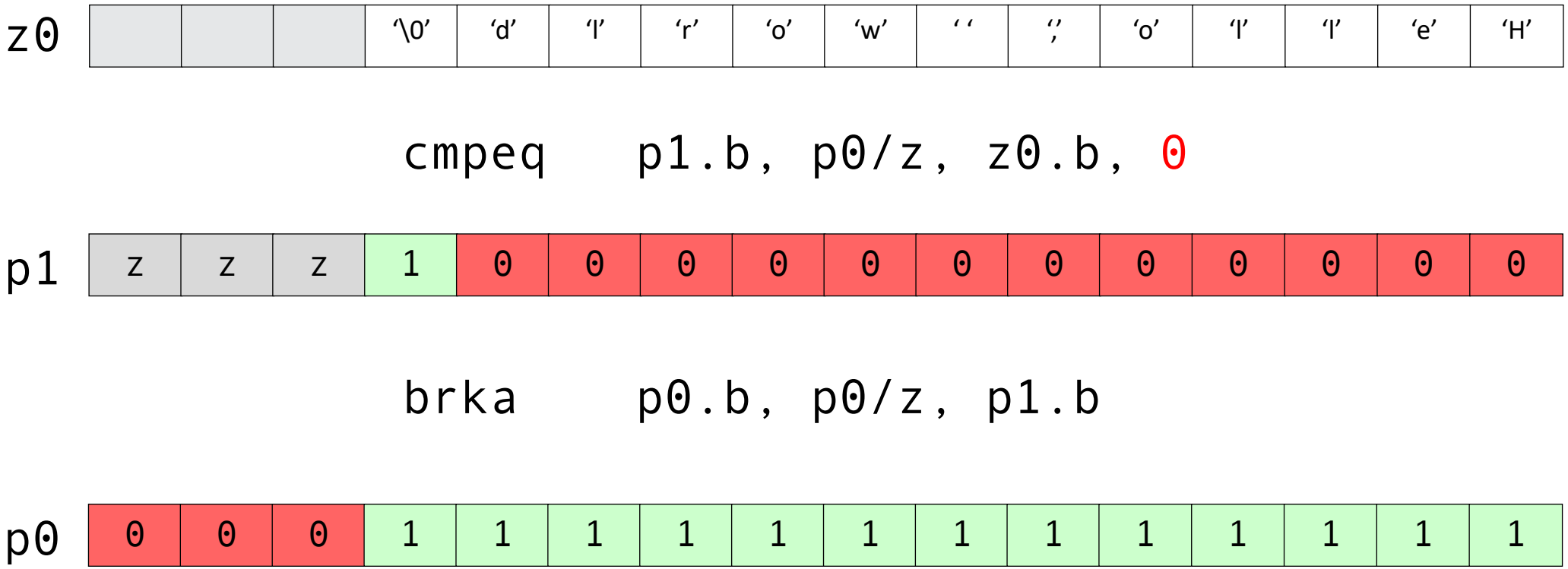
arm

Dealing with unknown loop bounds: Speculative Load



arm

Dealing with unknown loop bounds: Speculative Load



arm

Dealing with unknown loop bounds: Speculative Load

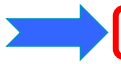
```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```

Store up to the first ‘\0’ (inclusive)

arm

Dealing with unknown loop bounds: Speculative Load

```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```



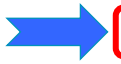
Increase x2 by the number of 1s in p0 (up to the '\0')

- Notice: by the end of the copy, x2 will be equal to the string length

arm

Dealing with unknown loop bounds: Speculative Load

```
sve_strcpy:
  mov     x2, 0
  ptrue   p2.b
loop:
  setffr
  ldff1b  z0.b, p2/z, [x1, x2]
  rdffr   p0.b, p2/z
  cmpeq   p1.b, p0/z, z0.b, 0
  brka    p0.b, p0/z, p1.b
  st1b    z0.b, p0, [x0, x2]
  incp    x2, p0.b
  b.none  loop
  ret
```



cmpeq set the flags; if no ‘\0’ was found, keep copying

arm

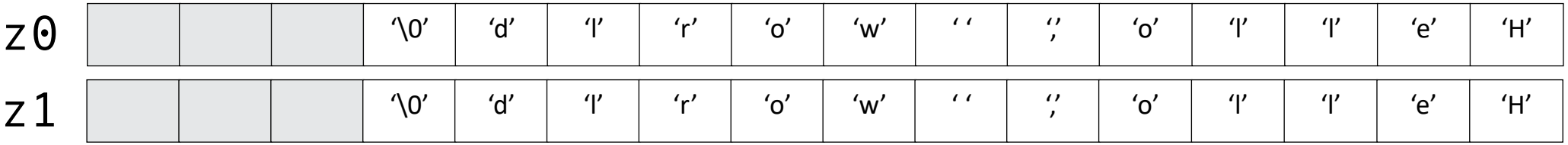
String comparison

```
int strcmp(const char *lhs, const char *rhs)
{
  while (*lhs == *rhs && *lhs != '\0')
    lhs++, rhs++;
  return (*lhs - *rhs);
}
```

String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison



arm

String comparison

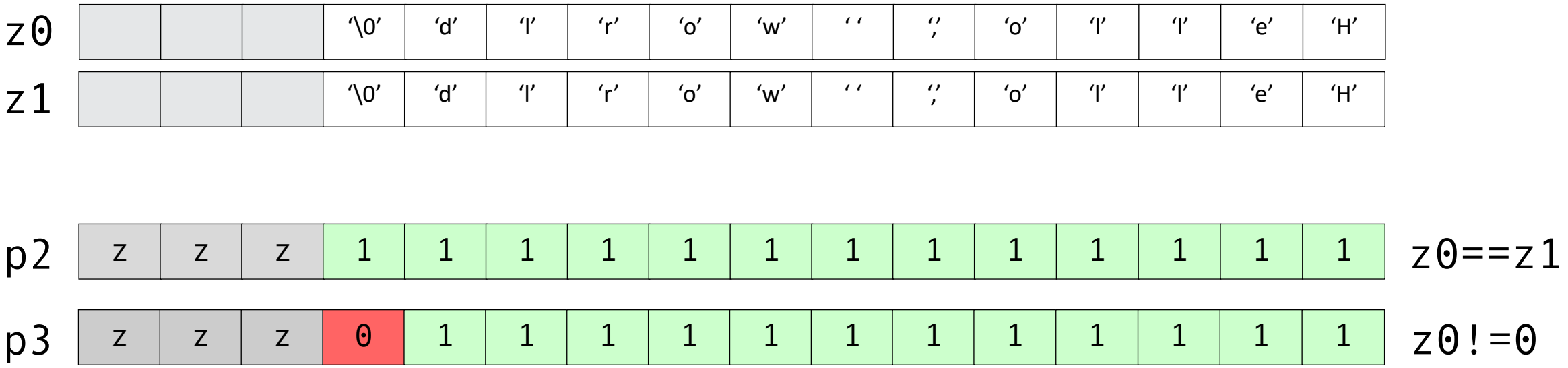
```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```


arm

String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison

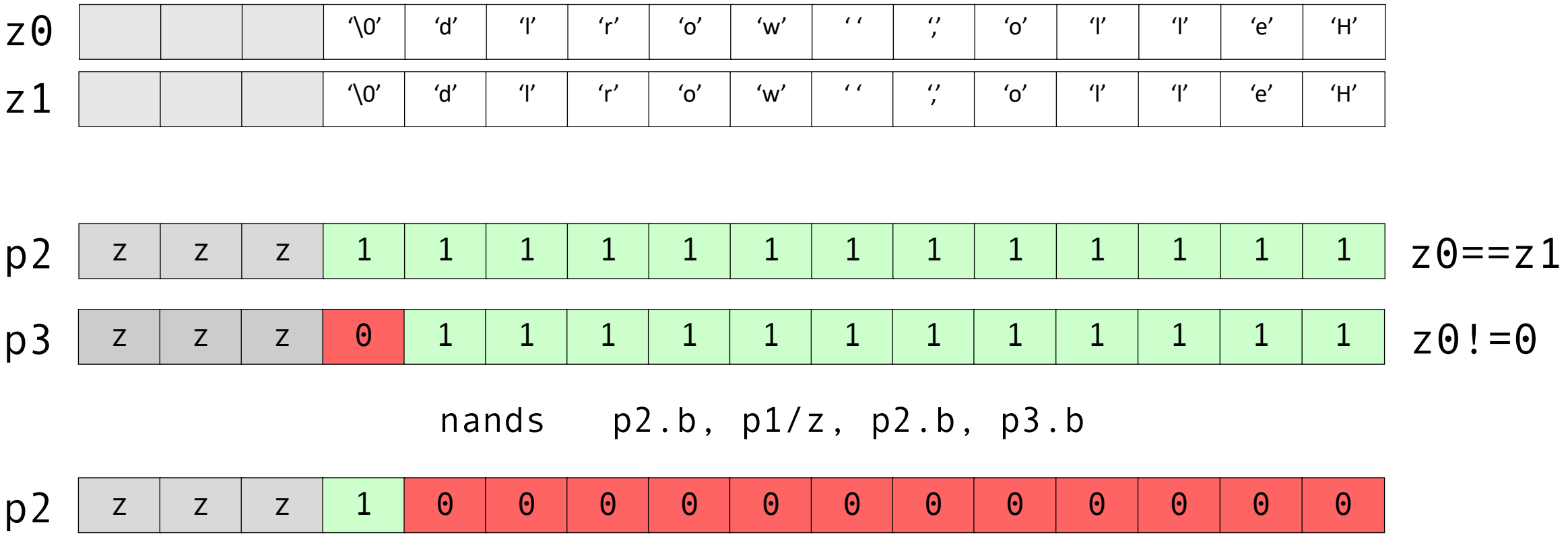


arm

String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison



arm

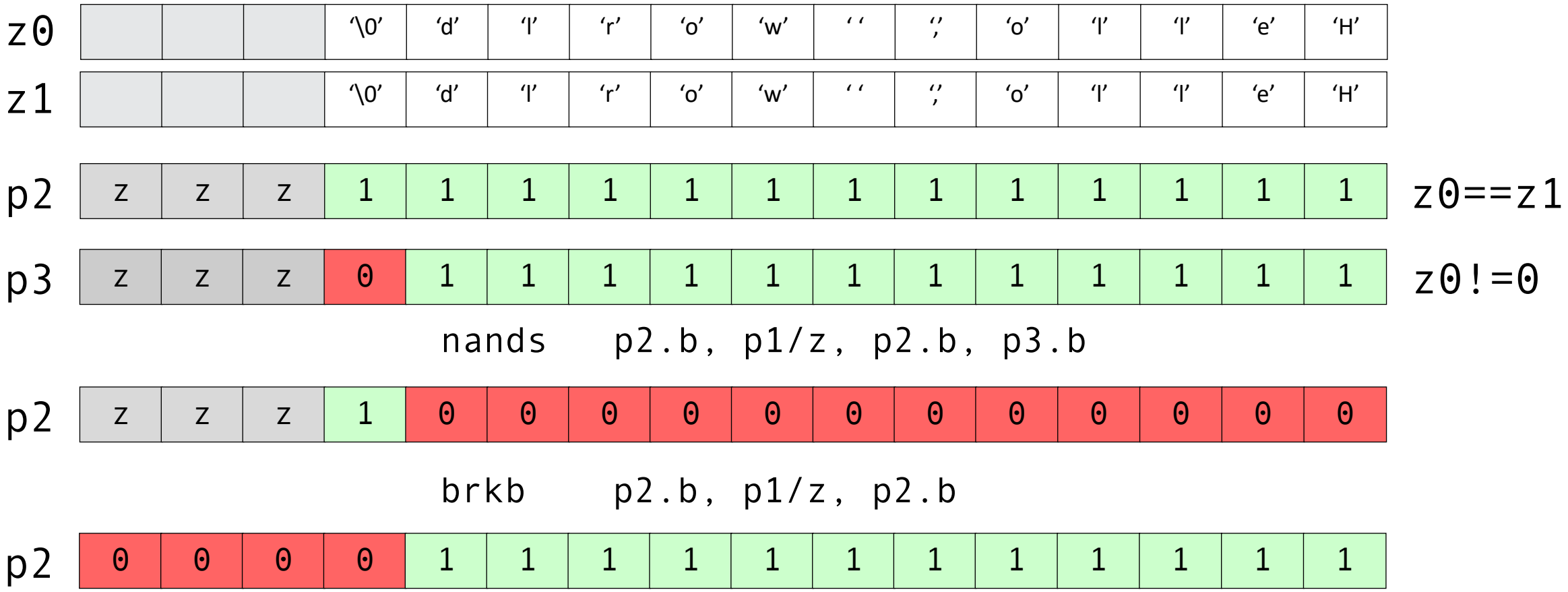
String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison

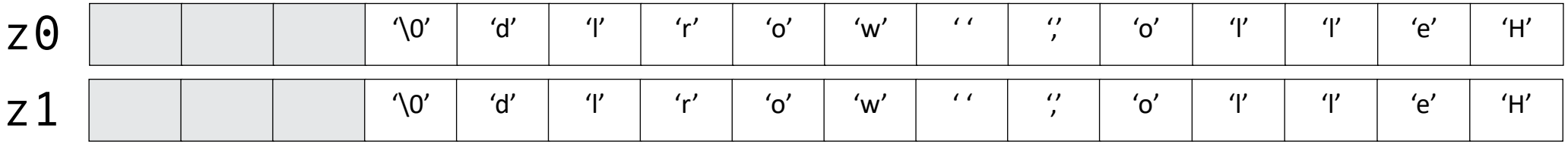


arm

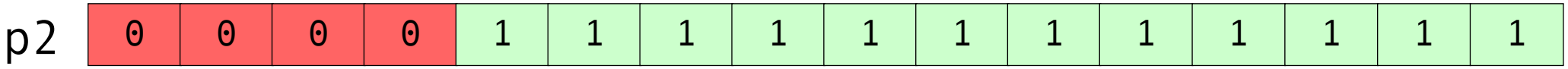
String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

String comparison

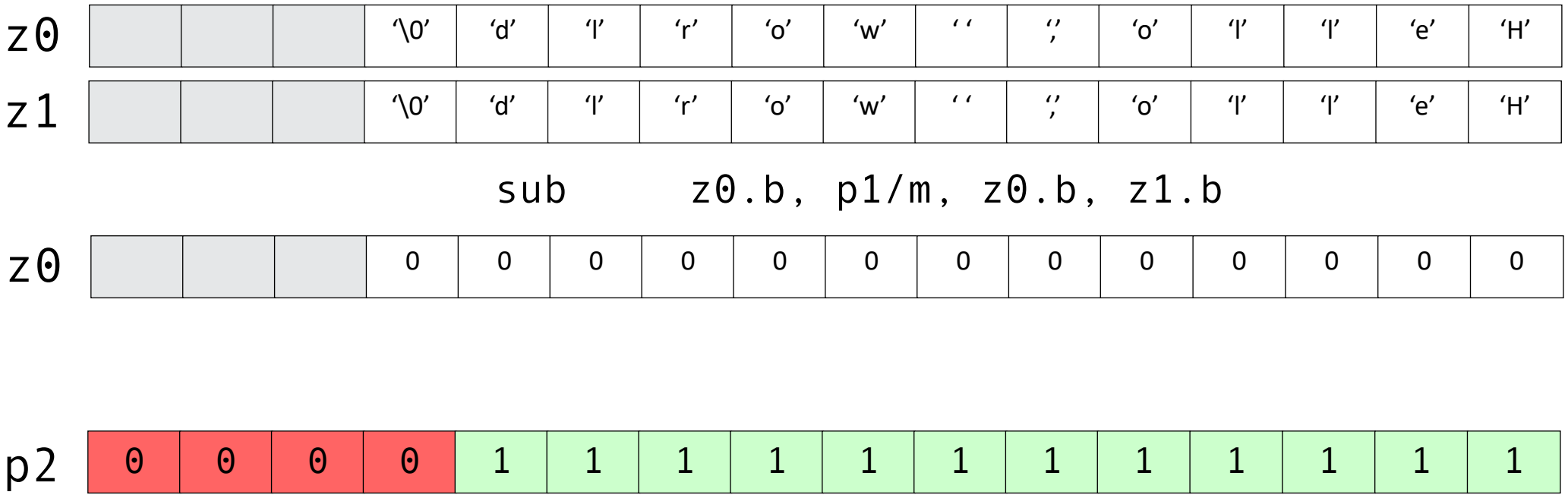



```
sub     z0.b, p1/m, z0.b, z1.b
```



arm

String comparison



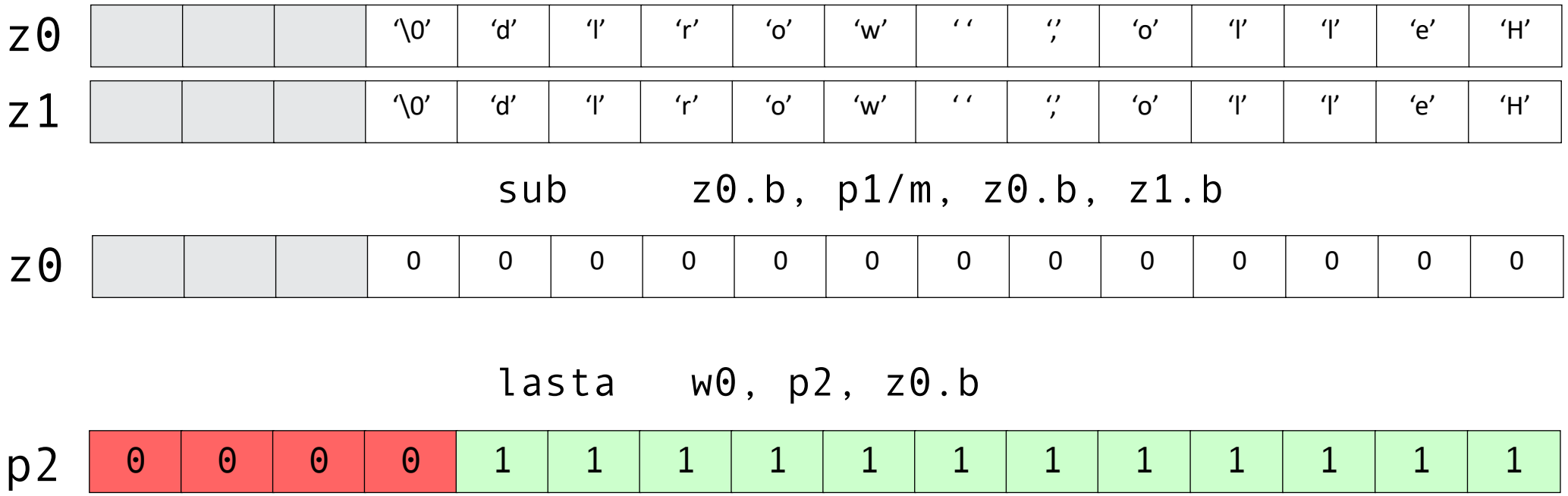
arm

String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

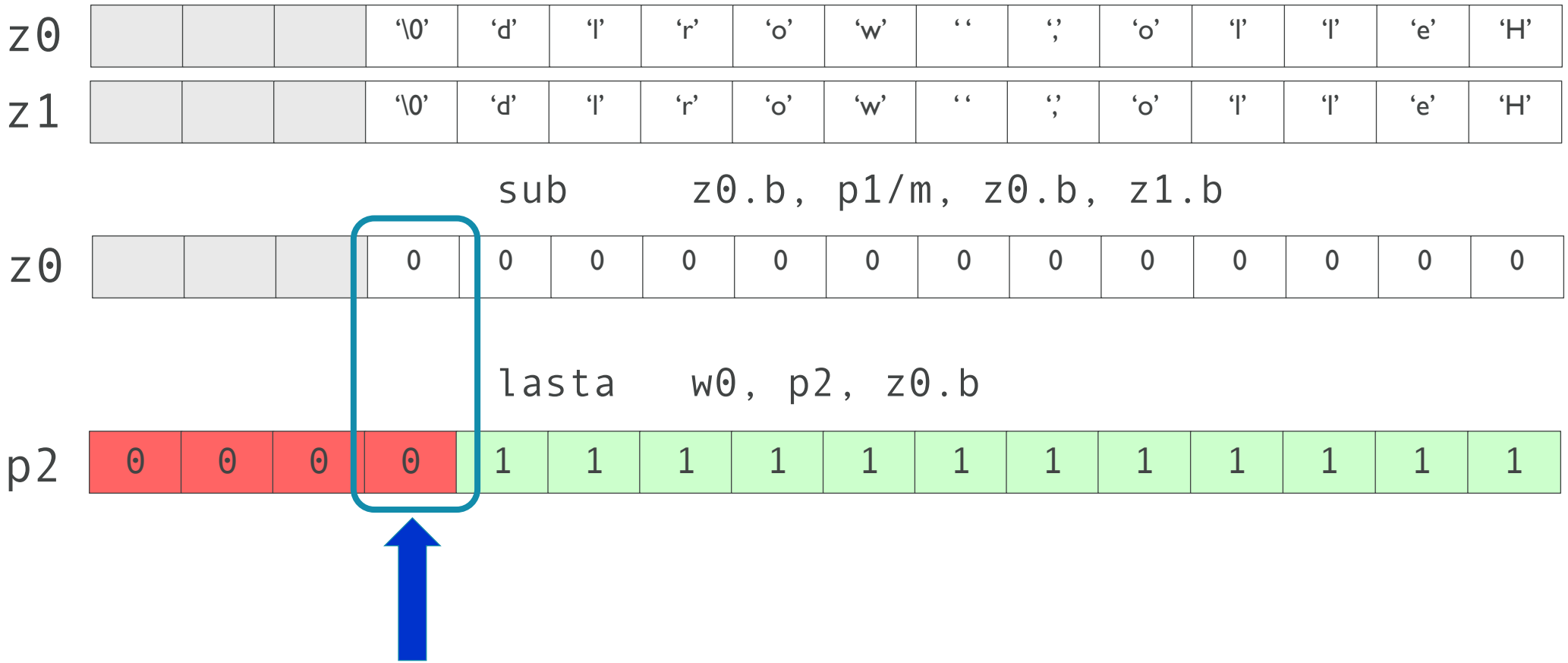

arm

String comparison



arm

String comparison



String comparison

```
sve_strcmp:
  mov x2, 0
  ptrue   p0.b
loop:
  setffr
  ldff1b  z0.b, p0/z, [x0, x2]
  ldff1b  z1.b, p0/z, [x1, x2]
  rdffr   p1.b, p0/z
  incp    x2, p1.b
  cmpeq   p2.b, p1/z, z0.b, z1.b
  cmpne   p3.b, p1/z, z0.b, 0
  nands   p2.b, p1/z, p2.b, p3.b
  b.none  loop
terminate:
  brkb    p2.b, p1/z, p2.b
  sub     z0.b, p1/m, z0.b, z1.b
  lasta   w0, p2, z0.b
  sxtb    w0, w0
  ret
```

arm