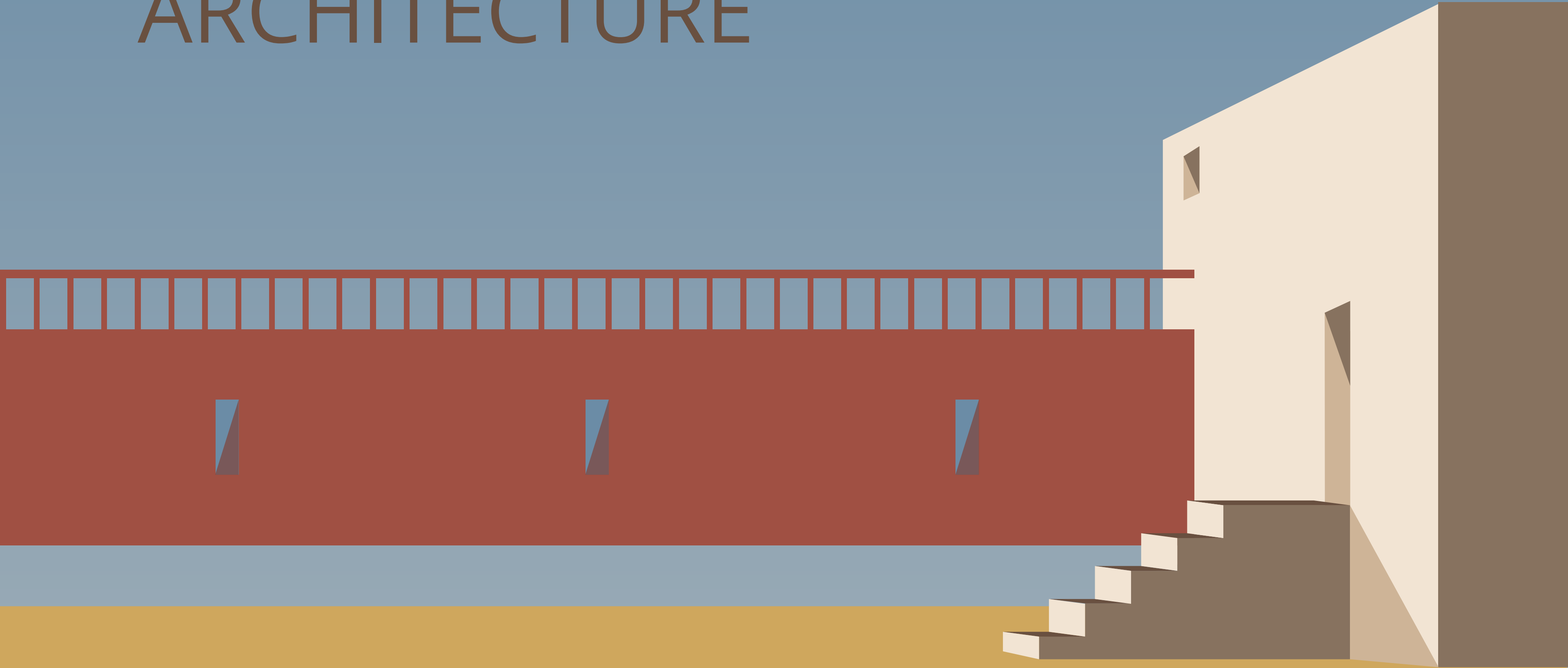
PARALLELISM IN ARCHITECTURE

(*real* architecture)



Vitruvius

Roman architect
Author of *De architectura, libri decem*
Importance of geometric principles in architectural design

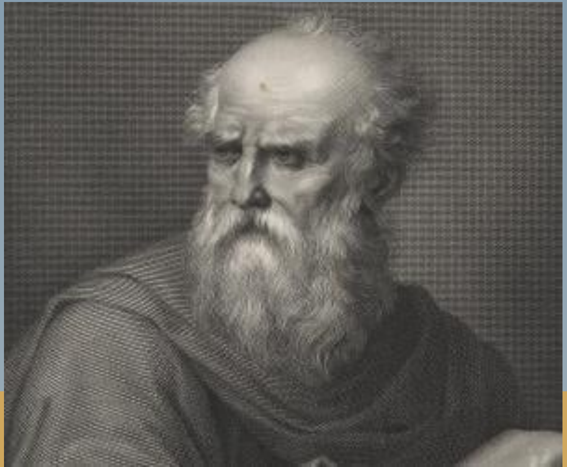


image source

Opus Quadratum

Squared-off blocks are placed in parallel

Alternating direction of placement improves **strength**

Different designs can be created, for **aesthetics**

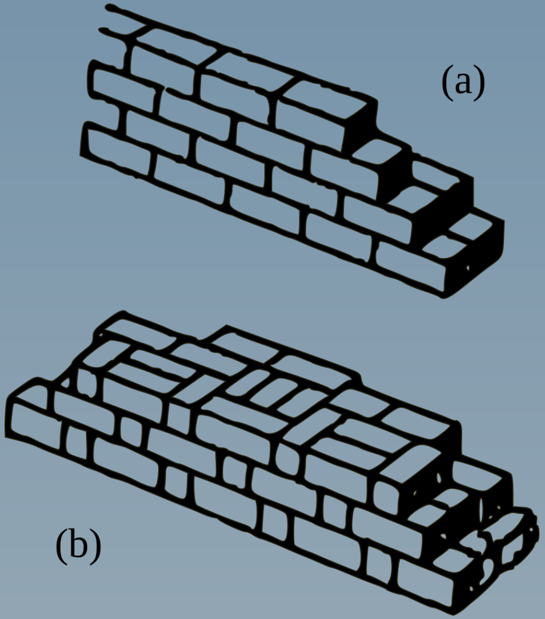


image source

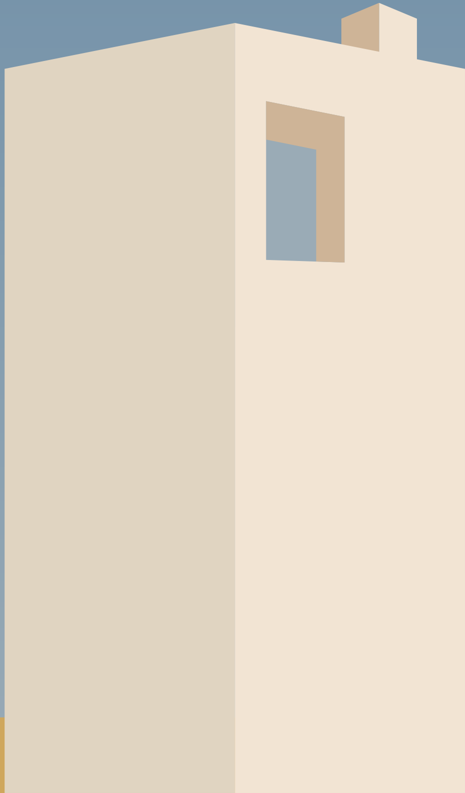
Contrast with parallelism in computer architecture

Used to improve performance (not aesthetics/strength)

We can’t really “see” it but spend all of our time designing for it

Comes from an observation about applications (rather than geometric principles):

- **Data-level parallelism** (DLP) means there are pieces of data that can be computed on at the same time
- **Task-level parallelism** (TLP) means there are independent tasks that can execute at the same time



Hardware support for parallelism

Instruction-level: running instructions at the same time – moderate exploitation of data-level parallelism

Thread-level: hardware support for switching between tasks

Hardware approaches for DLP: (such as vector machines and GPUs) apply single instruction to multiple pieces of data in parallel

Request-level: handling independent transactions (such as in servers, operating systems, databases)

Flynn’s taxonomy

Instruction stream

Data stream

	Single	Multiple
Single	SISD (More or less) what we’ve been studying so far	MISD Doesn’t really exist commercially
Multiple	SIMD Different data goes into FUs performing same operation	MIMD Independent processing units computing on independent data

???

What’s the point of exploiting DLP?
(In what applications might we be applying the same operation to different pieces of data?)

???

What advantages might SIMD-style computing have?

Vector architectures

Hail from the 60s, popular in the the supercomputers of the 70s (Cray)

Place data in *vector registers* for computation

- Cray-1 (1976): 8 vector registers of 64 values each

Vector loads/stores can be pipelined: amortize latency

image source

Vector instructions (RISC-V V extension)

Suffixes: .vv (vector-vector), .vx (vector-scalar), .vi (vector-immediate)

Arithmetic/logical/shift: vadd, vsub, vrsub, etc

Compare: vmseq, vmsne, vms{l,g}{t,e}[u]

Max/min: vmin[u], vmax[u]

Multiply-add (like dot product): vmacc, vnmsac, vmadd, vnmsub

Merge (set based on mask): vmerge

Reductions: vredsum, vredand, vredor, vredxor

source

Non-vector example of vectorizable code

Y = aX + Y

```
# assume X is in s1, Y is in s2
li s0, a # s0 = a
addi t0, s1, 256 # t0 = X + (64 * 4) (end address)
loop:
lw t1, 0(s1) # t1 = x[i]
mul t1, t1, s0 # x[i] = x[i] * a
lw t2, 0(s2) # t2 = y[i]
add t2, t2, t1 # y[i] = x[i] * a + y[i]
addi s1, s1, 4 # i++
addi s2, s2, 4 # i++
bne s1, t0, loop
```

Compare to vectorized version

Y = aX + Y

```
li s0, a
addi t0, s1, 256
loop:
lw t1, 0(s1)
mul t1, t1, s0
lw t2, 0(s2)
add t2, t2, t1
addi s1, s1, 4
addi s2, s2, 4
bne s1, t0, loop
```

```
li s0, a
vld v0, s1 # v0 = X
vld v1, s2 # v1 = Y
vmul.vx v0, v0, s0 # X = a * X
vadd.vv v1, v0, v1 # Y = a * X + Y
vst v1, s2
```

OR JUST:

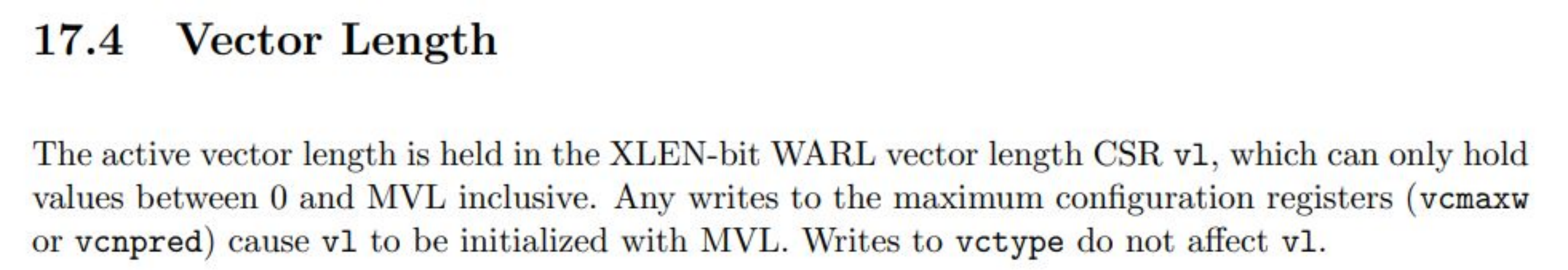
```
li s0, a
vld v0, s1
vld v1, s2
vmacc.vx v1, s0, v0 # Y = a * X + Y
vst v1, s2
```

What assumption are we making about our data here?

How to handle a loop like this?

```
for (int i = 0; i < 50; i++) {
    y[i] = a * x[i] + y[i];
}
```

```
setvl t0, s1
# vl, t0 = min(MVL, s1)
li s0, a
vld v0, s1
vld v1, s2
vmul.vx v0, v0, s0
vadd.vv v1, v0, v1
vst v1, s2
```



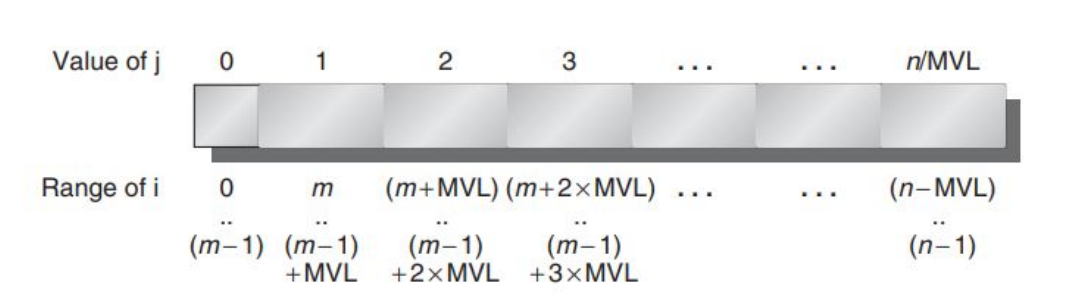
17.4 Vector Length

The active vector length is held in the XLEN-bit WARL vector length CSR `vl`, which can only hold values between 0 and MVL inclusive. Any writes to the maximum configuration registers (`vcmaxw` or `vcnpred`) cause `vl` to be initialized with MVL. Writes to `vctype` do not affect `vl`.

Strip mining

Compiler generating vectorized code when the # of loop iterations is unknown

H&P fig. 4.6



```
start = 0;
vlen = (n % MVL)
for (int j = 0; j < n / MVL; j++) {
    for (int i = start; i < start + vlen; i++) {
        y[i] = a * x[i] + y[i];
    }
    start += vlen;
    vlen = MVL;
}
```

this loop can be vectorized

What purpose does vmerge serve?

```
for (int i = 0; i < 64; i++) {
    if (x[i] != 0) {
        y[i] = a * x[i];
    }
}
```

What if we could tell the processor to only do this operation for indices i where x[i] != 0?

```
li s0, a
vld v0, s1
vld v1, s2
vmsne v2, v0, 0 # v2[i] = x[i] != 0 ? 1 : 0
vmul.vx v0, v0, s0 # x[i] = a * x[i]
vmerge v1, v1, v0, v2 # y[i] = v2[i] ? x[i] : y[i]
vst v1, s2
```

Big picture/themes in architecture

Vector extensions are a really good example of the interplay between ISA, hardware, and compiler

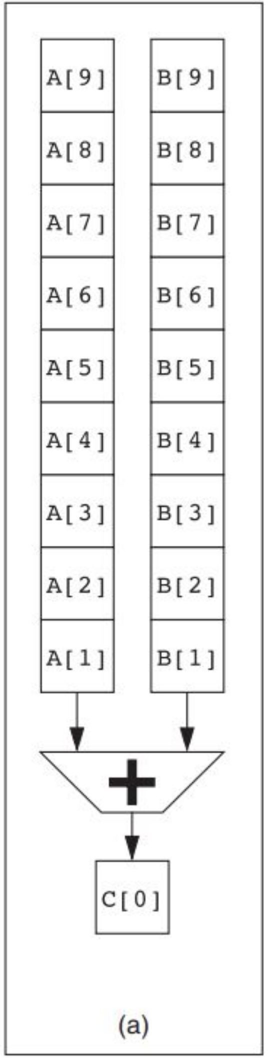
- How should the ISA support efficient hardware design?
- What operations does the ISA need to provide in order to facilitate vectorized compilation?

???

What else does the compiler need to do to detect and compile vectorizable code?

???

How many functional units does a uarch need to support vector instructions? How do those functional units behave?



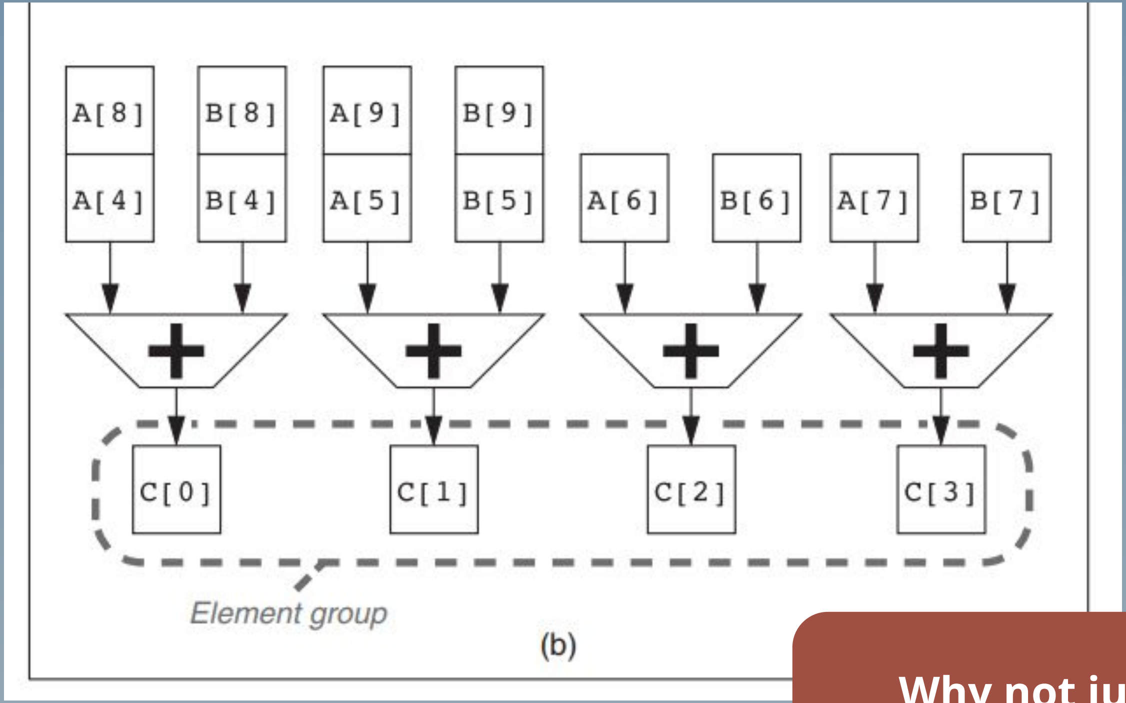
Simple approach: single pipelined FU

Upsides:

- Less hardware
- Smaller clock cycle
- One result/cycle
- Data within vector assumed independent: no hazards

H&P fig. 4.4

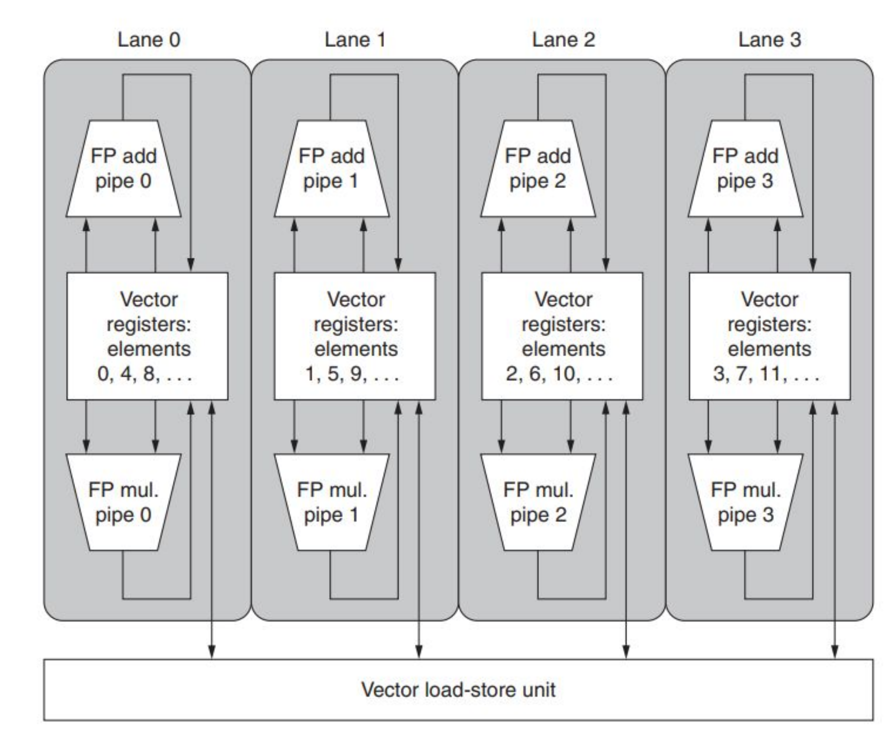
More efficient approach: multiple FUs



H&P fig. 4.4

Why not just have 64 unpipelined FUs?

Lanes

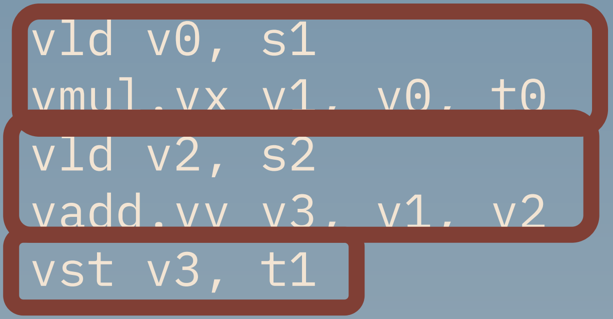

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

H&P fig. 4.5

Measuring performance: convoys and chimes

Convoy: set of vector instructions that can potentially execute together

Chime: time it takes to execute a convoy



```
vld v0, s1
vmul.vx v1, v0, t0
vld v2, s2
vadd.vv v3, v1, v2
vst v3, t1
```

Approximation of runtime for this vector machine: 3 * vlen
What complicates this metric?

Complication of memory access

How do we vectorize this code?

```
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
        A[i][j] = 0;
        for (int k = 0; k < 100; k++) {
            A[i][j] += B[i][k] * C[k][j]
        }
    }
}
```