Vector processors

Vector processors: summary so far

Place data in vector registers for computation

Run same operation on every element of a vector

Operations:

- Load and store between memory and vector register
- Set vector length
- Computations (math/logic, reductions, masked operations)

Today: what about the hardware?

What about these loops?

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

Masked/conditional instructions:

```
li s0, a
vle32.v v1, s1
vle32.v v2, s2
vmsne.vi v0, v1, 0
# v0[i] = x[i] != 0 ? 1 : 0
vmul.vx v2, v1, s0, v0.t
vse32.v v1, s2
```

```
for (int i = 0; i < n; i++) {
    X[m[i]] = X[m[i]] + Y[n[i]];
}
```

Gather: collect all valid X[m[i]], Y[n[i]] in smaller vectors

Scatter: put the data back into X[m[i]], Y[n[i]]

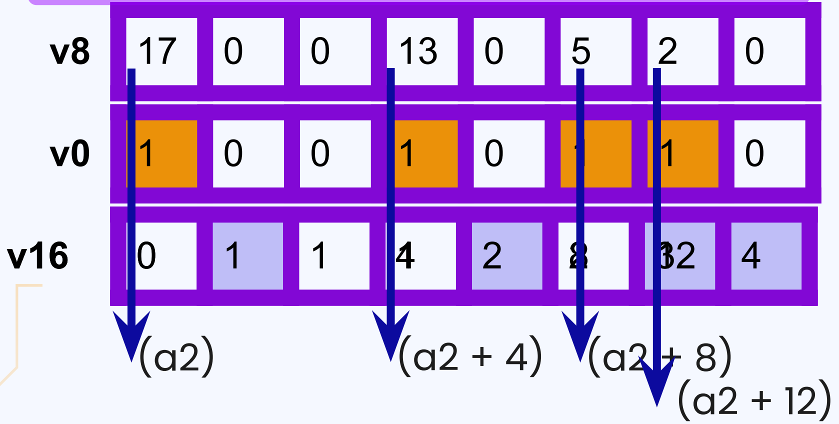
In RVV: indexed load/stores, also the `vrgather` instruction

Compact non-zero example (why?)

Page 82 of RVV spec (provided code also strip mines)

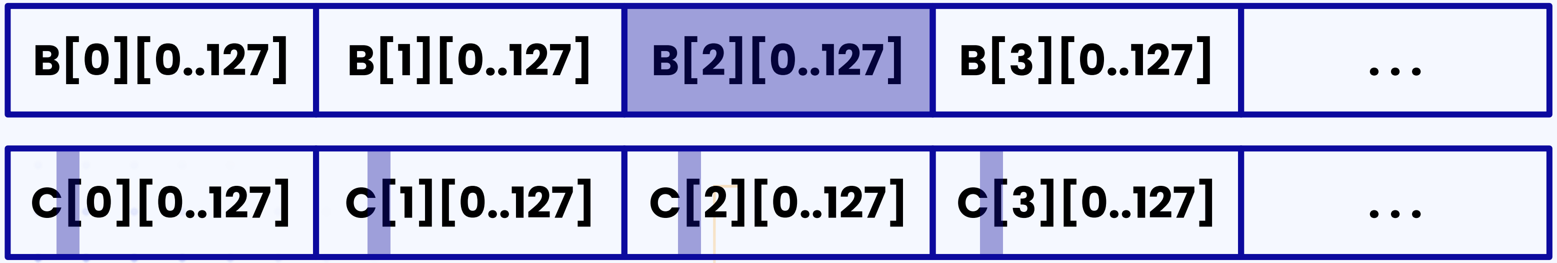
```
size_t compact_non_zero(
  size_t n,
  const int* in,
  int* out) {
    size_t i;
    size_t count = 0;
    int *p = out;
    for (i=0; i<n; i++) {
        const int v = *in++;
        if (v != 0)
            *p++ = v;
    }
    return (size_t) (p - out);
}
```

```
vle32.v v8, (a1)
vmsne.vi v0, v8, 0
vcpop.m a5, v0
viota.m v16, v0
vsll.vi v16, v16, 2, v0.t
vsuxei32.v v8, (a2), v16, v0.t
```

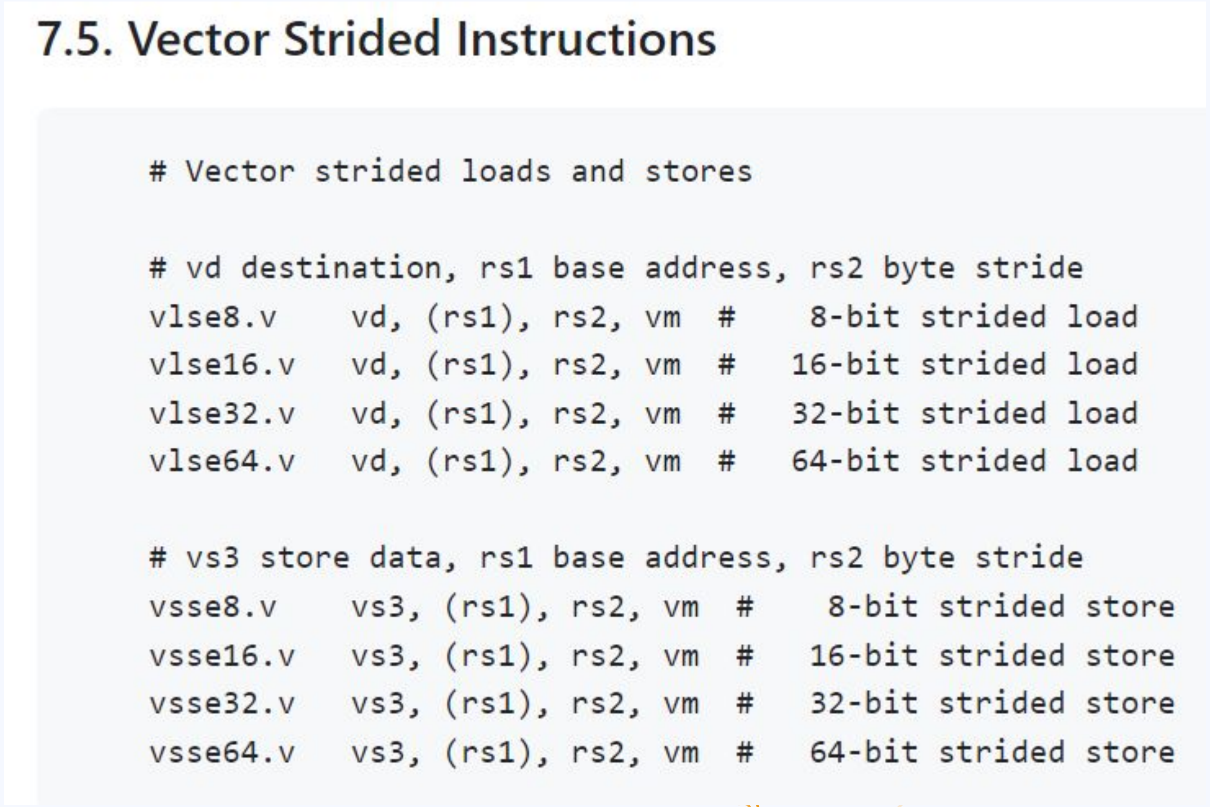


Matrix multiplies

```
for (int i = 0; i < 128; i++) {
    for (int j = 0; j < 128; j++) {
        A[i][j] = 0;
        for (int k = 0; k < 128; k++) {
            A[i][j] += B[i][k] * C[k][j]
        }
    }
}
```



Strided loads/stores



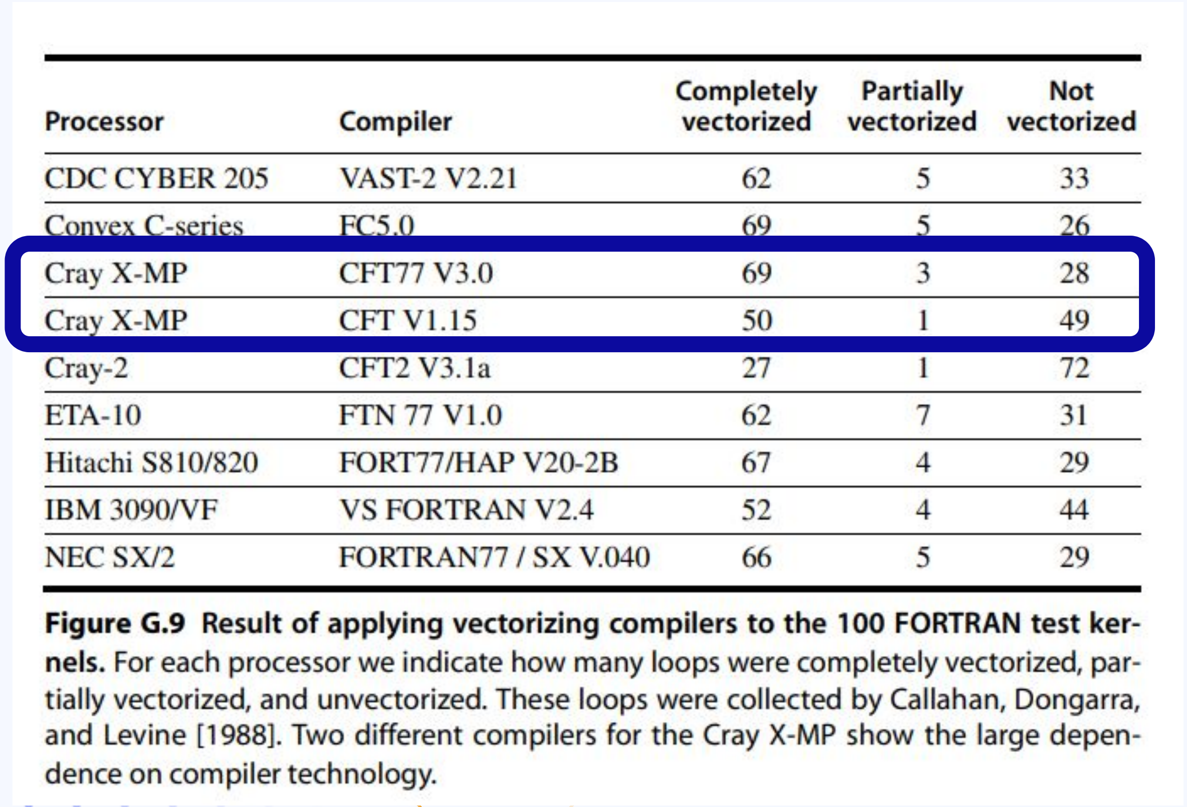
7.5. Vector Strided Instructions

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
vlse8.v    vd, (rs1), rs2, vm  #    8-bit strided load
vlse16.v   vd, (rs1), rs2, vm  #   16-bit strided load
vlse32.v   vd, (rs1), rs2, vm  #   32-bit strided load
vlse64.v   vd, (rs1), rs2, vm  #   64-bit strided load

# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v    vs3, (rs1), rs2, vm  #    8-bit strided store
vsse16.v   vs3, (rs1), rs2, vm  #   16-bit strided store
vsse32.v   vs3, (rs1), rs2, vm  #   32-bit strided store
vsse64.v   vs3, (rs1), rs2, vm  #   64-bit strided store
```

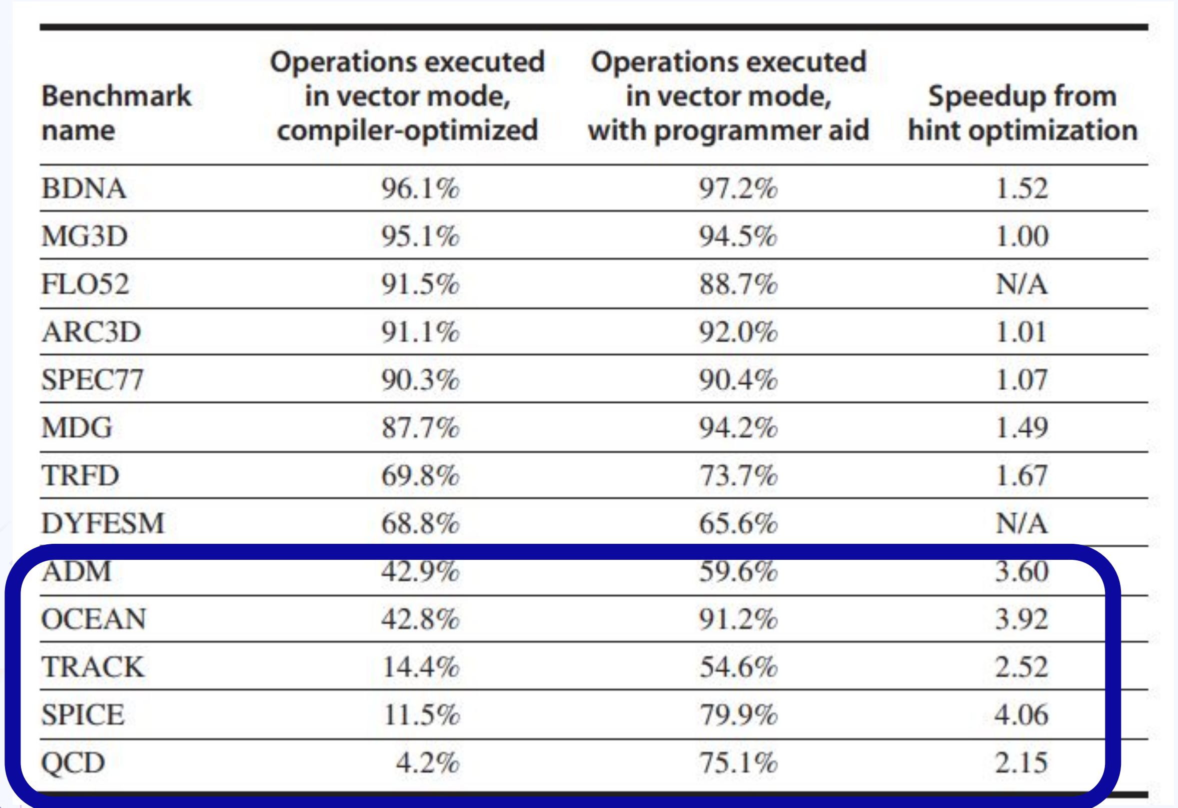
Compiler effectiveness



Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

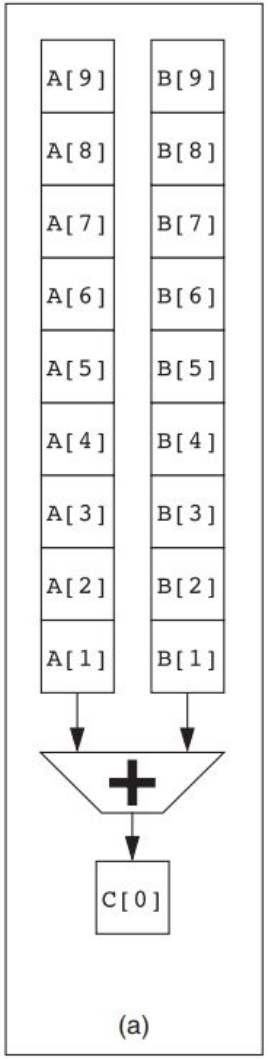
Figure G.9 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

Compiler🤝Programmer



Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

H&P Fig. 4.7

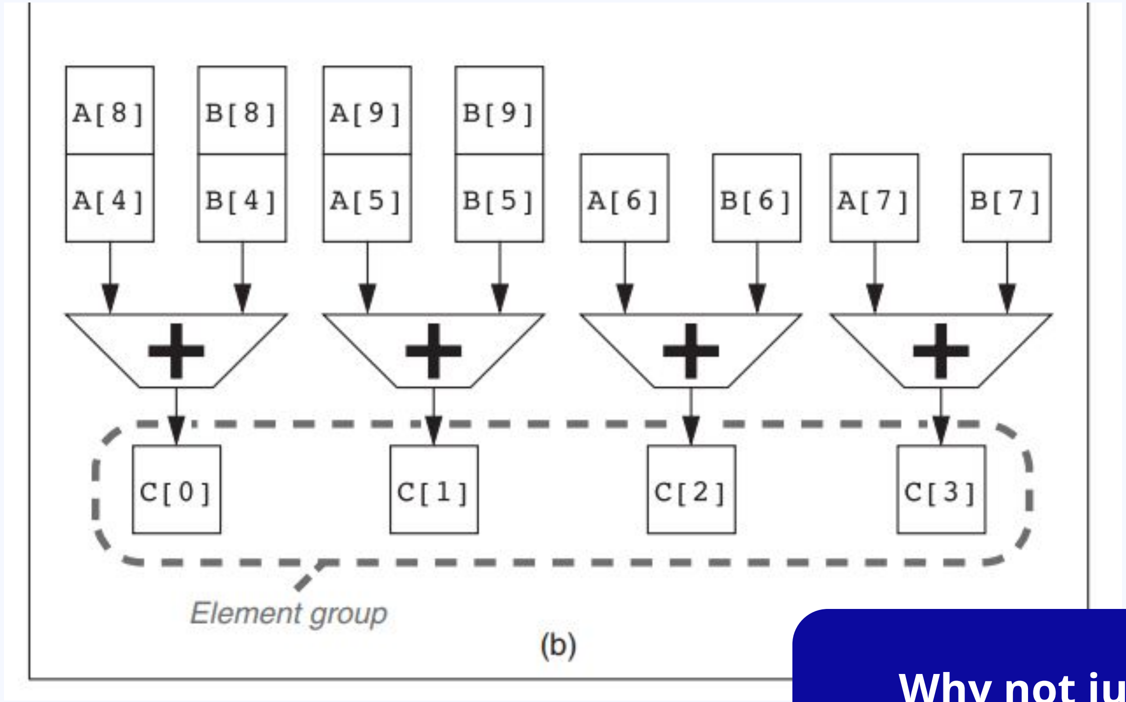


Simple approach: single pipelined FU

Upsides:

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

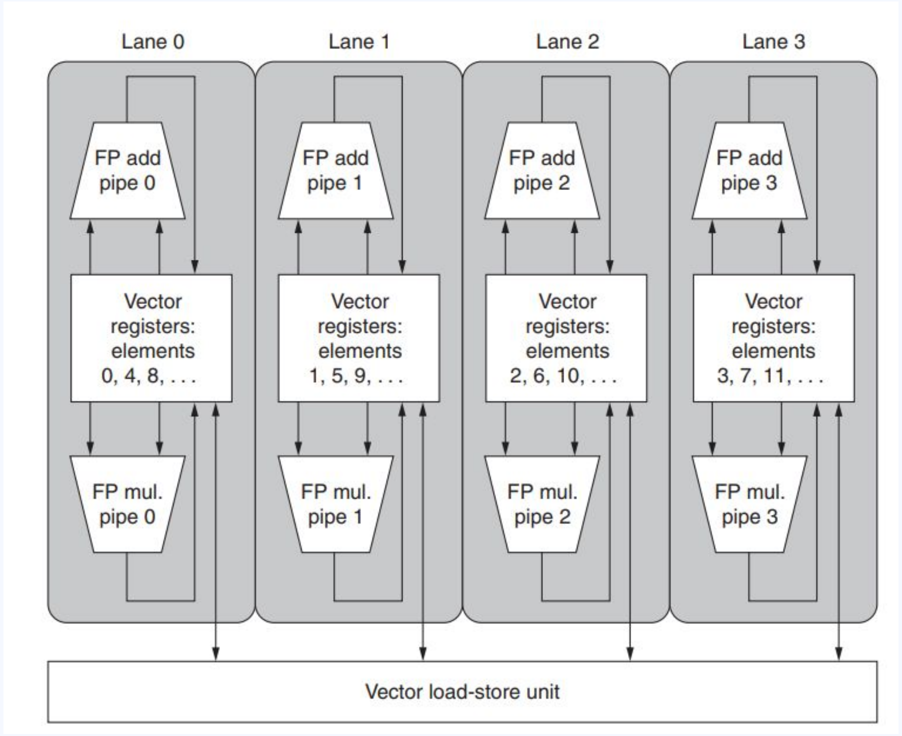
More efficient approach: multiple FUs



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

? ? ?

Theory: if we can start a load every cycle, eventually, we get a throughput of 1 piece of data/cycle
Practice: how do we start a load every cycle if loads take multiple cycles?

Memory banks

How do strided accesses complicate the advantages of memory banks?

Measuring performance: convoys and chimes

Convoy: set of vector instructions that can potentially execute together

Chime: time it takes to execute a convoy

```
vle32.v v0, s1
vmul.vx v1, v0, t0
vle32.v v2, s2
vadd.vv v3, v1, v2
vse32.v v3, t1
```

Approximation of runtime for this vector machine: 3 chimes (~3 * vlen/lanes clock cycles)
What complicates this metric?

Measuring performance: chaining

What does it look like to execute this code w/ one load/store unit and one ALU/mul unit?

```
vle32.v v0, s1
vmul.vx v1, v0, t0
vle32.v v2, s2
vadd.vv v3, v1, v2
vse32.v v3, t1
```

Startup, dead time

Amdahl’s law

Used to assess theoretical effectiveness of speedup

In a nutshell: gains in speeding up a portion of a program are limited by the fraction of time that portion is actually used

Mathematically:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

For parallelization: serial bottleneck (non-parallelizable code) limits effectiveness of vector processors

Cray-1 Architecture

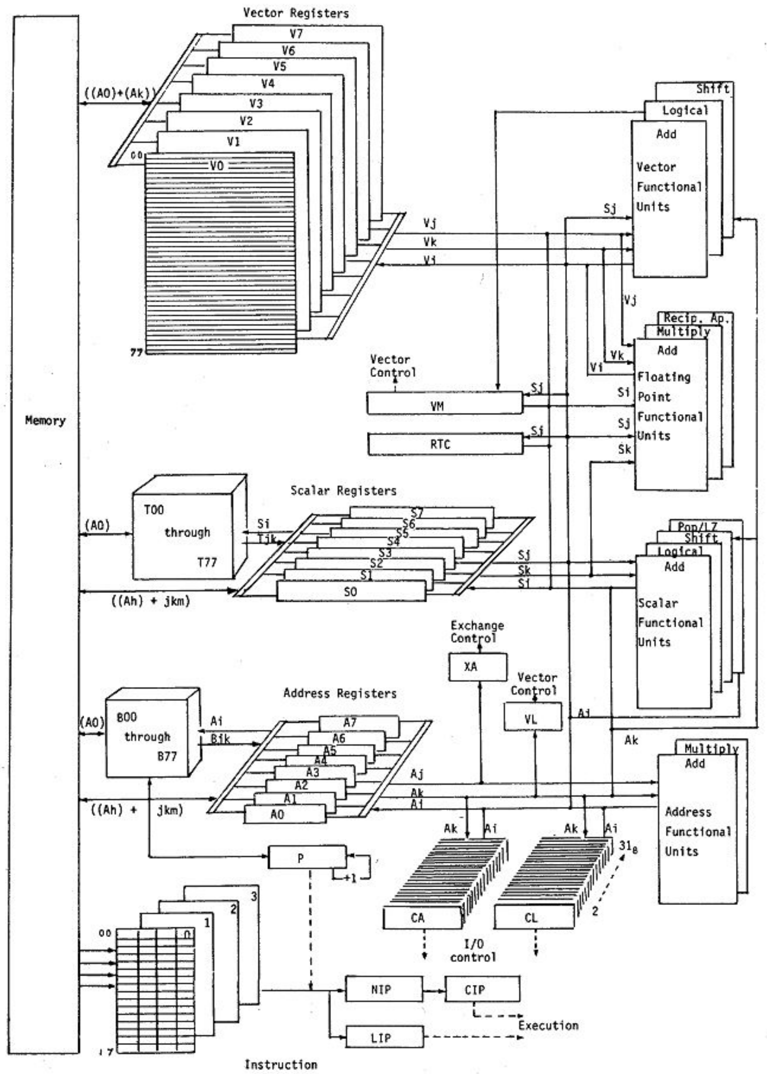


image source