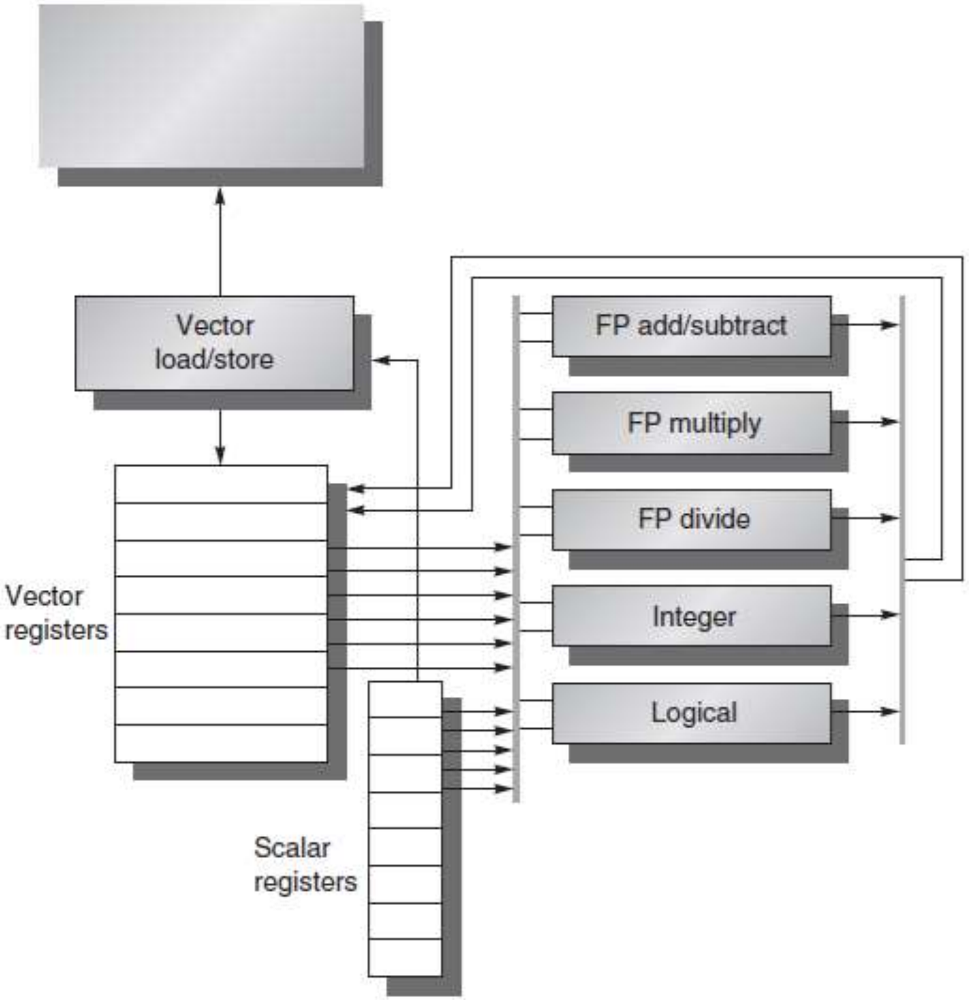
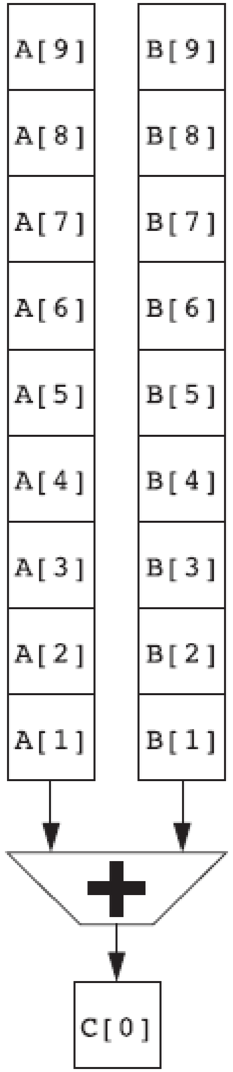
VMIPS


Vector
load/store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector
registers
Scalar
registers


A[9]
A[8]
A[7]
A[6]
A[5]
A[4]
A[3]
A[2]
A[1]
B[9]
B[8]
B[7]
B[6]
B[5]
B[4]
B[3]
B[2]
B[1]
+
C[0]

VMIPS ISA

- **Vector-Vector (VV):** Operations between two vector registers
- **Vector-Scalar (VS):** Operations between a vector register and a scalar value
- **Memory operations:** Load/store entire vectors or with stride/indexed addressing

Instruction	Operands	Function
ADDVV.D ADDVS.D	V1,V2,V3 V1,V2,F0	Add elements of V2 and V3, then put each result in V1. Add F0 to each element of V2, then put each result in V1.
SUBVV.D SUBVS.D SUBSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtract elements of V3 from V2, then put each result in V1. Subtract F0 from elements of V2, then put each result in V1. Subtract elements of V2 from F0, then put each result in V1.
MULVV.D MULVS.D	V1,V2,V3 V1,V2,F0	Multiply elements of V2 and V3, then put each result in V1. Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D DIVVS.D DIVSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Divide elements of V2 by V3, then put each result in V1. Divide elements of V2 by F0, then put each result in V1. Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

DAXPY Example

DAXPY: $Y = a \times X + Y$

MIPS (Scalar)

```
        L.D     F0,a          ;load scalar a
        DADDIU  R4,Rx,#512    ;last address to load
Loop:   L.D     F2,0(Rx)      ;load X[i]
        MUL.D   F2,F2,F0      ;a × X[i]
        L.D     F4,0(Ry)      ;load Y[i]
        ADD.D   F4,F4,F2      ;a × X[i] + Y[i]
        S.D     F4,9(Ry)      ;store into Y[i]
        DADDIU  Rx,Rx,#8      ;increment index to X
        DADDIU  Ry,Ry,#8      ;increment index to Y
        DSUBU   R20,R4,Rx     ;compute bound
        BNEZ    R20,Loop      ;check if done
```

VMIPS (Vector)

```
L.D      F0,a        ;load scalar a
LV       V1,Rx       ;load vector X
MULVS.D  V2,V1,F0    ;vector-scalar multiply
LV       V3,Ry       ;load vector Y
ADDVV.D  V4,V2,V3    ;add vectors
SV       V4,Ry       ;store the result
```

Vector code: 6 instructions vs. ~600 for scalar (for 64 elements)

Pipeline stalls occur once per vector instruction rather than once per element

Vector Execution Time

- Convoy approximation
  - **Convoy**: a set of vector instructions that can execute together without structural hazards (resource constraints, dependency, etc.)
  - **Chime**: the approximate time to execute one convoy
- Chaining
  - Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - The results from the first functional unit in the chain are “forwarded” to the second functional unit

Example

```
L.D      F0,a         ;load scalar a
LV       V1,Rx        ;load vector X
MULVS.D  V2,V1,F0     ;vector-scalar multiply
LV       V3,Ry        ;load vector Y
ADDVV.D  V4,V2,V3     ;add vectors
SV       V4,Ry        ;store the result
```

- 3 Convoys (single load/store)
  - Convoy 1: LV, MULVS.D (by chaining)
  - Convoy 2: LV, ADDVV.D (by chaining)
  - Convoy 3: SV
- It takes 3 chimes
- For instance, with 64-element vectors, it takes 3 * 64 = 192 cycles