ILP vs DLP

ILP = Instruction Level Parallelism
DLP = Data Level Parallelism

---

ILP

Vector Procs can also exploit ILP to a limited extent, it is just more effective to exploit arbitrary ILP on OOO and VLIW Proc

OOO and VLIW Exploit ILP when arbitrary instructions can be executed in parallel

---

DLP

Vector Exploits DLP where the same instruction operating on different data is executed in parallel

OOO and VLIW can definitely still exploit DLP, it is just more efficient to exploit DLP on Vector Proc
**SUBWORD - SIMD**

ADD A NEW INSTRUCTION THAT DOES FOUR 8B ADDITIONS IN PARALLEL

- **ADDU.Q6** rd, r+, rs

  \[
  R[rd][7:0] = R[r+][7:0] + R[rs][7:0] \\
  \]

**SYNTAX**

\[
\text{FOR } i = 0; i < n; i++ \]

\[
C[i] = A[i] + B[i]
\]

**ASSUME** A, B, C ARE ARRAYS OF BYTES NOT ARRAYS OF WORDS

**SCALAR ASSEMBLY**

- **Loop:**
  - lw
  - lw
  - add
  - sb
  - add
  - add
  - add
  - add
  - beq

**SUBWORD - SIMD ASSEMBLY**

- **Loop:**
  - lw
  - lw
  - addu.q6
  - sw
  - addu
  - addu
  - addu
  - addu
  - beq

**Four Adds in Parallel**

- **Load/Store Byte**
  - Pointer bump by 1 byte

**SUBWORD - SIMD MICROARCHITECTURE**

- Simply break carry chain
  - Can also do subword multiply and floating point
VECTOR INSTRUCTION SET ARCHITECTURE

- ADD A NEW "VECTOR REGISTER FILE" THAT HOLDS VECTOR INSTEAD OF SCALAR VALUES
- ADD NEW "VECTOR INSTRUCTIONS" THAT OPERATE ON THE VECTOR REGISTERS
- NUMBER OF ELEMENTS IN A VECTOR REGISTER IS CALLED THE "HARDWARE VECTOR LENGTH"

VECTOR ARCHITECTURAL STATE

<table>
<thead>
<tr>
<th>SCALAR RF</th>
<th>VECTOR RF</th>
<th>VECTOR REGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>vro</td>
<td>f0</td>
</tr>
<tr>
<td>r1</td>
<td>vr1</td>
<td>f1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>r31</td>
<td>vr31</td>
<td>f4</td>
</tr>
</tbody>
</table>

VECTOR INSTRUCTIONS

- VECTOR-VECTOR MUL
  \[ \text{MUL.VV } \text{vrA}, \text{vrS}, \phi, \text{vrS1} \]
  
  for \( i = 0 \) to \( \text{VLR}-1 \)
  
  \[ \text{VR[vrA]}[i] = \text{VR[vs]}[i] \times \text{VR[vs1]}[i] \]

- VECTOR-SCALAR MUL
  \[ \text{MUL.VS } \text{vrA}, \text{vs}, \text{rs} \]
  
  for \( i = 0 \) to \( \text{VLR}-1 \)
  
  \[ \text{VR[vrA]}[i] = \text{VR[vs]}[i] \times \text{R[rs]} \]
- **Vector Unit-Stride Load**

  \[ \text{LW.V} \ vrd, \ rb \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ \text{VR}[\text{vrd}][i] = M[R[rb] + 4 \times i] \]

- **Vector Strided Load**

  \[ \text{LWST.V} \ vrd, \ rb, \ rs \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ \text{VR}[\text{vrd}][i] = M[R[rb] + 12 \times rs \times i] \]

- **Vector Indexed Load**

  \[ \text{LWX.V} \ vrd, \ rb, \ rs, \ vrs \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ \text{VR}[\text{vrd}][i] = M[R[rb] + \text{VR}[\text{vrs}]] \]

- **Vector Unit-Stride Store**

  \[ \text{SW.V} \ vrs, \ rb \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ M[R[rb] + 4 \times i] = \text{VR}[\text{vrs}][i] \]

- **Set Less Than (Flag Reg)**

  \[ \text{SLLT.f} \ f d, \ vrs_f, \ vrs_l \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ \text{FR}[\text{fd}][i] = (\ \text{VR}[\text{vrs}_f][i] < \text{VR}[\text{vrs}_l][i]) \]

- **Vector-Vector Multiply (Unpred Mask)**

  \[ \text{MUL.VV} \ vrd, \ vrs \_f, \ vrs \_l, \ f \_s \]

  \[ \text{for } i = 0 \to \text{VLR}-1 \]

  \[ \text{if (FR[f_s][i] == 1)} \]

  \[ \text{VR}[\text{vrd}][i] = \text{VR}[\text{vrs}_f][i] \times \text{VR}[\text{vrs}_l][i] \]
- SET ACTIVE VECTOR LENGTH
  
  SETVL r0, r6

  \[ temp = \text{min}(\text{mmovw}, r[r6]) \]
  
  \[ r[r0] = temp; \text{vlen} = temp \]

VECTOR CODE EXAMPLES

```
for (int i = 0; i < n; i++)
  B[i] = A[i] * c
```

<table>
<thead>
<tr>
<th>Scalar Assembly</th>
<th>Vector Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td>LW r1, 0(r2)</td>
<td>LW* v, vr1, r2</td>
</tr>
<tr>
<td>MUL r3, r1, r4</td>
<td>MUL* vs, vr2, vr1, r4</td>
</tr>
<tr>
<td>SW r3, 0(r5)</td>
<td>SW* v, vr2, r5</td>
</tr>
<tr>
<td>ADDU r2, r2, 4</td>
<td></td>
</tr>
<tr>
<td>ADDU r3, r3, 4</td>
<td></td>
</tr>
<tr>
<td>ADDU r7, r7, -1</td>
<td></td>
</tr>
<tr>
<td>BNE r7, r0, loop</td>
<td></td>
</tr>
</tbody>
</table>

If \( n = 2 \) and \( \text{mmovw} = 4 \)

```
ADDU r8, r0, 2
SETVL r9, r8
LW* vr1, r2
MUL* vs, vr2, vr1, r4
SW* v, vr2, r5
```

Even though \text{mmovw} \text{vector length} is four, because we used \text{setvl} to set the \text{active vector length}, we will only process two elements per vector instruction.

If \( n \) or \( \text{mmovw} \) is unknown at compile time, must use \text{stripmining}, execute in \( \text{mmovw} \) "chunks"
for (int i = 0; i < n; i++)
B[i] = A[i] * C

**SCALAR ASSEMBLY**

**VECTOR ASSEMBLY**

```c
loop:
  lw  r1, 0(r2)
  mul  r3, r1, r4
  sw  r3, 0(r5)
  addu r2, r2, 4
  addu r3, r3, 1
  addu r7, r7, -1
  beq r7, r6, loop
```

```c
loop:
  setvl rb, r7
  lw  v1, r1, r2
  mul.v v2, v1, r4
  sw  v2, r2, r5
  slc  r9, rb, r2
  addu r2, r2, r9
  addu r5, r5, r9
  subu r7, r7, r8
  bne  r7, r6, loop
```

---

**EXECUTING MORE IRREGULAR DATA-LEVEL PARALLELISM**

**LESS STRUCTURE IN DATA ACCESS**

```c
for (i = 0; i < n; i++)
  C[i] = A[i] + B[i]

for (i = 0; i < n; i++)
  C[i] = x * A[i] + D[2*i]
```

```c
for (i = 0; i < n; i++)
  E[C[i]] = D[A[i]] + D[i]
```

**LESS STRUCTURE IN CONTROL FLOW**

```c
for (i = 0; i < n; i++)
  if (A[i] < 0)
    x = y
  else
    x = z
  C[i] = x * A[i] + D[i]
```

```c
for (i = 0; i < n; i++)
  if (A[B[i]] < 0)
    C[i] = D[i] + E[i]
```

---
Using vector indexed accesses (scatter/gather) to implement irregular memory accesses

```
for (i = 0; i < n; i++)
    A[i] = B[C[i]]
```

Using vector flags/masks to implement irregular control flow

```
for (i = 0; i < n; i++)
    if (C[i] < 0)
        B[i] = A[i] * C
```

**VECTOR MICROARCHITECTURE**

In li and sl stages we can now also request to read on write a vector of data. Separate load/store pipes will eventually allow multiple vector instructions to execute in parallel.

Assume 32 vector registers, vector register file has a total of 32 x MUIV elements

\[ \text{VLIR} = \text{VLOAD mem response buf} \]
\[ \text{VSIR} = \text{VSTORE mem req buf} \]
Executing Vector Instructions

HWLEN = 2
MUL.b.8
F D R 0 y l y W
R 0 y l y y W

HWLEN = 4
MUL.b.8
F D R 0 y l y y W
R 0 y l y y y W
R 0 y l y y y y W
R 0 y l y y y y y W

HWLEN = 8
MUL.b.8
F D R 0 y l y y W
R 0 y l y y y W
R 0 y l y y y y W
R 0 y l y y y y y W
R 0 y l y y y y y y W
R 0 y l y y y y y y y W
R 0 y l y y y y y y y y W

HWLEN = 4
LW.b.8
F D R 10 Li W
R 10 Li W
R 10 Li W
R 10 Li W

IMPORTANT: WE READ BASE ADDRESS, GENERATE EFFECTIVE ADDRESSES, AND SEND OUT A REQUEST TO READ A VECTOR OF DATA INTO VLMR WHEN HANDLING FIRST ELEMENT.

2nd - 4th elements do nothing in R and do 50 stages. We just write back the appropriate element in the destination vector register.

HWLEN = 4
SW.b.8
F D R 1031 W
R 1031 W
R 1031 W
R 1031 W

IMPORTANT: WE READ BASE ADDRESS, GENERATE EFFECTIVE ADDRESSES, AND STORE THIS ADDRESS IN THE VSMR WHEN HANDLING THE FIRST ELEMENT.

All elements read store data and write VSMR. Last element sends out memory request to write a vector of data.
Let's study a simple loop executing on this vector microarchitecture.

**Scalar Assembly**

```
loop:
  lw      r1, 0(r2)
  mvl     r3, r1, r4
  sw      r3, 0(r5)
  addiu   r2, r2, 1
  addiu   r3, r3, 1
  addiu   r7, r7, -1
  bne     r2, r4, loop
```

**Vector Assembly**

```
loop:
  setvl   r8, r7
  lwv     vr1, r2
  mvlv   vr3, vr1, r4
  lwv     vr5, r5
  slv     r9, r8, 2
  adduv   r2, r2, r9
  adduv   r5, r5, r9
  subv   r7, r7, r8
  bne     r7, r8, loop
```

For reference, what is the execution time of scalar code?

```
LW     F D I m0(m1) m2 m3 W
MUL    F D I i0 i1 i2 i3 m0 m1 m2 m3 W
SW     F D D D D D D I xo xi yl y3 W
ADDIU  F F F F F F F F I xo xi yl y3 W
ADDIU  F F F F F F F F I xo xi yl y3 W
BNE    F D I xo xi yl y3 W
op A   F D I
op B   F D I
op C   F D I
LW
```

14 cycles

Assume G4 instructions

\# instm = 4 \times 64 = 448

cycles/instm = (14 \times 64) / 448 = 2

cycle time = 1 ns

**Total execution time** = 896 ns
**Basic Execution, \( \text{HWLEN} = 4 \)**

- **LW.V**
  - \( \text{F D L 0 U} \)
  - \( \text{L 0 U} \) via vector register file
  - \( \text{F 0 U} \)

- **MUL.VS**
  - \( \text{F D D D D D D D} \)
  - \( \text{R 0 Y 1 Y 1 Y 1} \)
  - \( \text{Y 1 Y 1 Y 1} \)

- **SW.V**
  - \( \text{F F F F F F E D D D D} \)
  - \( \text{N 0 S 1} \)

---

**Chaining, \( \text{HWLEN} = 4 \)**

- **LW.V**
  - \( \text{F D R 0 U} \)
  - \( \text{L 0 U} \) via vector file, analogous to scalar bypassing
  - \( \text{F 0 U} \)

- **MUL.VS**
  - \( \text{F D D D D D R 0 Y 1 Y 1 Y 1} \)
  - \( \text{Y 1 Y 1 Y 1} \)

- **SW.V**
  - \( \text{F F F E D D D D D D} \)
  - \( \text{N 0 S 1 U} \)

Stall due to \( \text{N 0 S 1 U} \) with port conflict

---

**Chaining, \( \text{HWLEN} = 2 \) (Assume \( n = 2 \))**

- **LW.V**
  - \( \text{F D R 0 U} \)
  - \( \text{L 0 U} \) via bypass network, stall in D

- **MUL.VS**
  - \( \text{F D D 0 Y 1 Y 1 Y 1} \)

- **SW.V**
  - \( \text{F D D D D D D 0 S 1 U} \)

Cannot hide ALU latency with DL
Casing, HWUL = B (Assume n = B)

DATA FOR FIRST ELEMENT IS READY EASILY!

Key Observation: With relatively little fetch issue bandwidth + multiple fetch/issue, we can still keep functional units busy for many cycles.

* Add more vregfile ports to exploit vector insta. level parallelism.

To keep all functional units busy at same time, need:

- Y-Pipe 2r 1w
- X-Pipe 2r 1w
- L-Pipe 1r 1w
- S-Pipe 2r 0w

Total: 2r 3w
CHAINING, WLEN = 8 (ASSUME n=8) W FROM VME BUS

I Vecow wifh Excel
AT SAME TIME, EQUIVALENT TO N/IQ SCALAR OPERATIONS
IN FLIGHT AT ONCE wI
SINGLE ISSUE!
# insta = 9 x 16 = 144

cycle/instr = (16 x 16) / 144 = 1.78

cycle time = 1/n

total execution time = 256 cycles

Speed up of 3.5 but needed to add 4x registers and 7 or 3w ports

What if M = 8?
MULTIPLE LANES

LANE 0

LANE 1

V Register File is partitioned by LANE.
Assume $\text{LANE} = 8 \rightarrow 2$ Lanes

only needs for \text{L}$ case
16 cycles/lmem, same as #1 lane if vecr = 4

Scalar code getting in the way
- # vec
- Decouple scalar processor
Multiple lanes and decoupled control proc.

Control processor is decoupled from vector unit.

Word vector command queue.