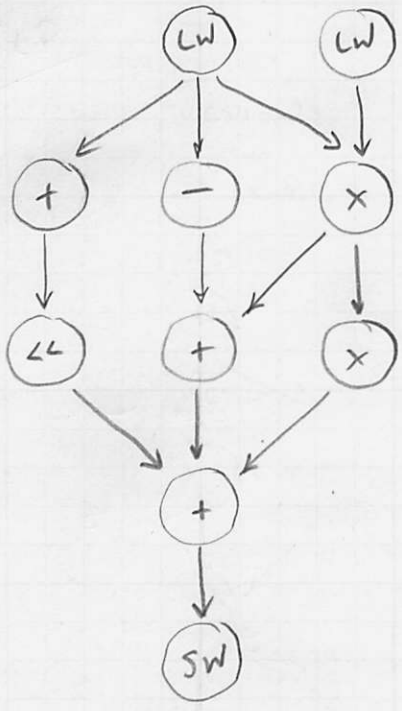
ILP vs DLP

ILP = INSTRUCTION LEVEL PARALLELISM

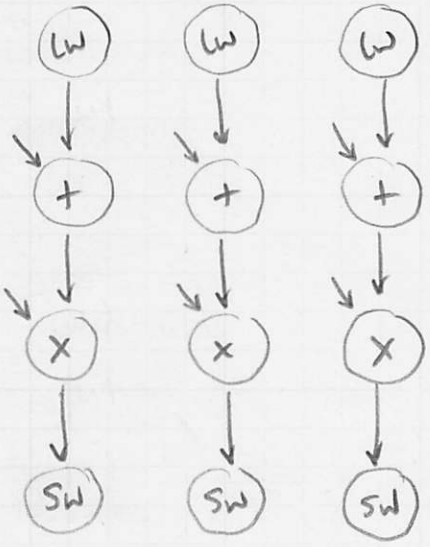
DLP = DATA LEVEL PARALLELISM



ILP

OOO AND VLIW EXPLOIT ILP WHERE ARBITRARY INSTRUCTIONS CAN BE EXECUTED IN PARALLEL

VECTOR PROCS CAN ALSO EXPLOIT ILP TO A LIMITED EXTENT, IT IS JUST MORE effective TO EXPLOIT ARBITRARY ILP ON OOO AND VLIW PROC



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.qb rd, rt, rs

```
R[rd][7:0]   = R[rt][7:0]   + R[rs][7:0]
R[rd][15:8]  = R[rt][15:8]  + R[rs][15:8]
R[rd][23:16] = R[rt][23:16] + R[rs][23:16]
R[rd][31:24] = R[rt][31:24] + R[rs][31:24]
```

SEMANTICS

```
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:
  lbu   r1, 0(r2)
  lbu   r3, 0(r4)
  ADDU  r5, r1, r3
  sb    r5, 0(r6)
  ADDIU r2, r2, 1
  ADDIU r4, r4, 1
  ADDIU r6, r6, 1
  ADDIU r7, r7, -1
  bne   r7, r0, loop
```

LOAD/STORE BYTE

Pointer bump by 1 BYTE

SUBWORD-SIMD ASSEMBLY

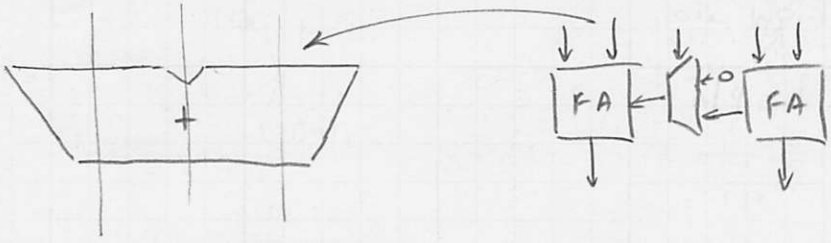
```
loop:
  lw      r1, 0(r2)
  lw      r3, 0(r4)
  ADDU.qb r5, r1, r3
  sw      r5, 0(r6)
  ADDIU   r2, r2, 4
  ADDIU   r4, r4, 4
  ADDIU   r6, r6, 4
  ADDIU   r7, r7, -1
  bne     r7, r0, loop
```

FOUR ADDS IN PARALLEL

LOAD/STORE WORD

ONLY n/4 ITERATIONS

SUBWORD-SIMD MICROARCHITECTURE



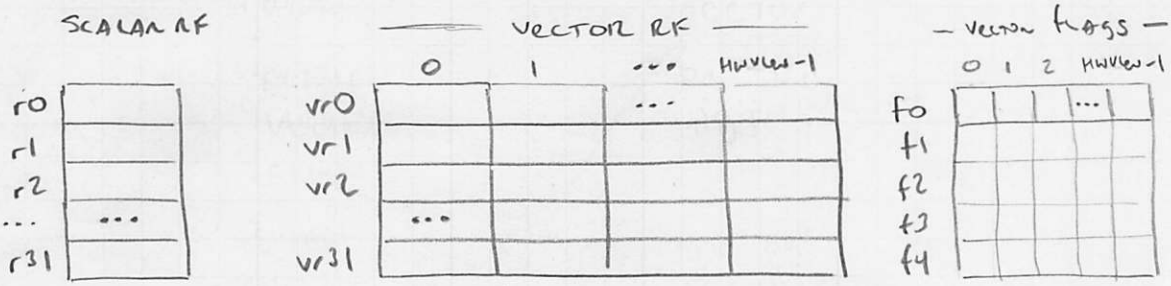
SIMPLY BREAK CARRY CHAIN

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"

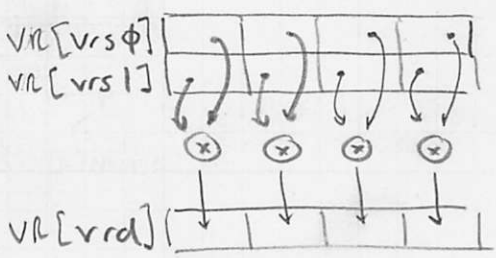
VECTOR ARCHITECTURAL STATE



VECTOR INSTRUCTIONS

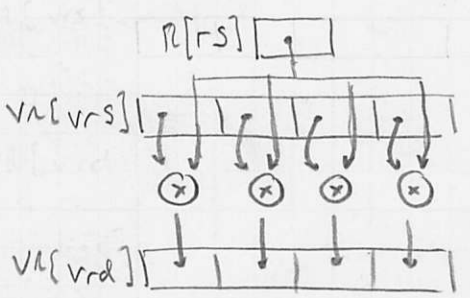
- VECTOR-VECTOR MULTIPLY

```
MUL.VV vrd, vrs0, vrs1

for i=0 to VLR-1
  VR[vrd][i] = VR[vrs0][i]
             x VR[vrs1][i]
```

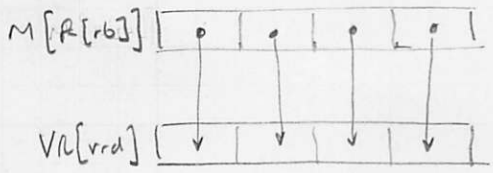


- VECTOR-SCALAR MULTIPLY

```
MUL.VS vrd, vrs, rs

for i=0 to VLR-1
  VR[vrd][i] = VR[vrs][i] x R[rs]
```

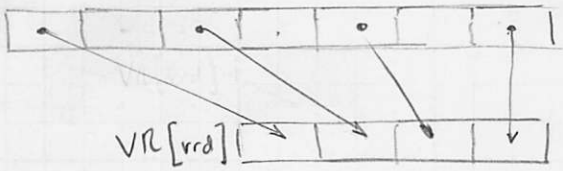


- VECTOR UNIT-STRIDE LOAD



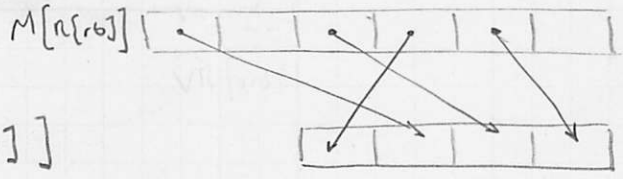

```
LW.V vrd, rb

for i=0 to VLR-1
  VR[vrd][i] = M[ R[rb] + 4xi ]
```

- VECTOR STRIDED LOAD



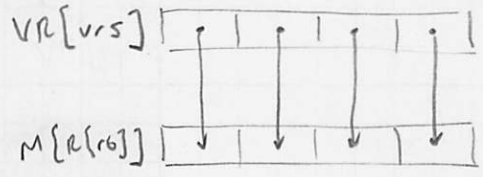

```
LWST.V vrd, rb, rs

for i=0 to VLR-1
  VR[vrd][i] = M[ R[rb] + R[rs]xi ]
```

- VECTOR INDEXED LOAD




```
LWX.V vrd, rb, vrs

for i=0 to VLR-1
  VR[vrd][i] = M[ R[rb] + VR[vrs] ]
```

- VECTOR UNIT-STRIDE STORE




```
SW.V vrs, rb

for i =0 to VLR-1
  M[ R[rb] + 4 x i ] = VR[vrs][i]
```

- SET LESS THAN (FLAG REG)

```
SLT.f.vv fd, vrs0, vrs1

for i =0 to VLR-1
  FR[fd][i] = ( VR[vrs0][i] < VR[vrs1][i] )
```

- VECTOR-VECTOR MULTIPLY (UNDER MASK)

```
MUL.VV vrd, vrs0, vrs1, fs

for i = 0 to VLR-1
  if ( FR[fs][i] == 1 )
    VR[vrd][i] = VR[vrs0][i] x VR[vrs1][i]
```

- SET ACTIVE VECTOR LENGTH

SETVL ra, rb

temp = MIN(HWVLEN, R[rb])
R[ra] = temp ; VLR = temp

VECTOR CODE EXAMPLES

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

SCALAR ASSEMBLY

```
loop:
  LW    r1, 0(r2)
  MUL   r3, r1, r4
  SW    r3, 0(r5)
  ADDIU r2, r2, 4
  ADDIU r5, r5, 4
  ADDIU r7, r7, -1
  BNE   r7, r0, loop
```

VECTOR ASSEMBLY

```
  LW.V    vr1, r2
  MUL.VS  vr2, vr1, r4
  SW.V    vr2, r5
```

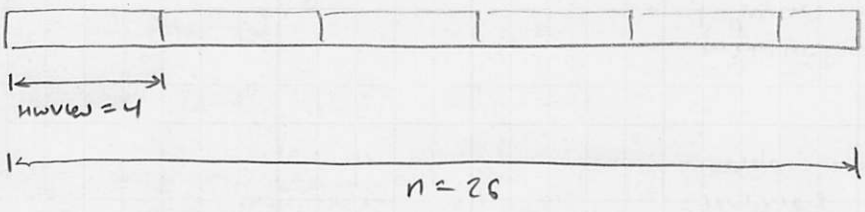
THIS ASSUMES n == HWVLEN !
WHAT IF HWVLEN != n ?

if n = 2 AND HWVLEN = 4

```
  ADDIU r8, r0, 2
  SETVL r9, r8
  LW.V    vr1, r2
  MUL.VS  vr2, vr1, r4
  SW.V    vr2, r5
```

} EVEN THOUGH HARDWARE VECTOR LENGTH IS FOUR, BECAUSE WE USED SETVL TO SET THE ACTIVE VECTOR LENGTH WE WILL ONLY PROCESS TWO ELEMENTS PER VECTOR INSTRUCTION

IF n OR HWVLEN IS UNKNOWN AT COMPILE TIME MUST USE STRIPMINING, EXECUTE IN HWVLEN "CHUNKS"



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

SCALAR ASSEMBLY

```
loop:
  lw    r1, 0(r2)
  MUL   r3, r1, r4
  SW    r3, 0(r5)
  ADDIU r2, r2, 4
  ADDIU r5, r5, 4
  ADDIU r7, r7, -1
  BNE   r7, r0, loop
```

VECTOR ASSEMBLY

INITIALIZES TO n

```
loop:
  SETVL  r8, r7
  LW.V   vr1, r2
  MUL.VV vr2, vr1, r4
  SW.V   vr2, r5
  SLL    r9, r8, 2      # MULT VLR x4
  ADDU   r2, r2, r9     # Pointers + VLR
  ADDV   r5, r5, r9     #
  SUBU   r7, r7, r8     # i = i - VLR
  BNE    r7, r0, loop
```

VECTOR CODE

EXECUTING MORE IRREGULAR DATA-LEVEL PARALLELISM

LESS STRUCTURE IN DATA ACCESS →

LESS STRUCTURE IN CONTROL FLOW ↓

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

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

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

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

```
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] ]
```

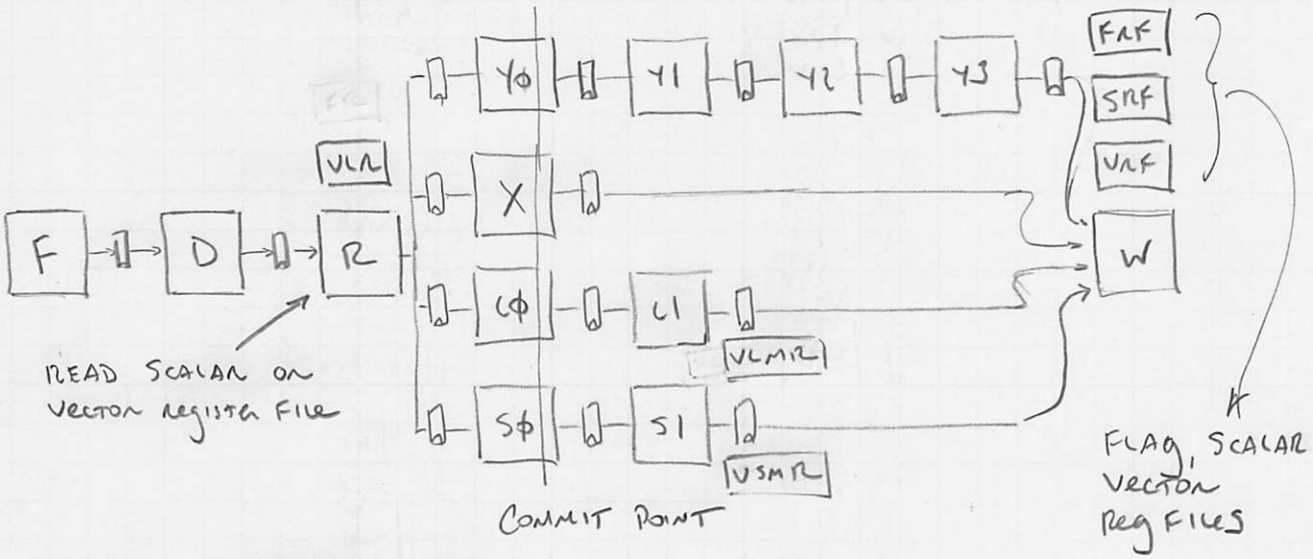
```
lw.v   vr1, r2      <- c ptr
lwx.v  vr2, r3, vr1 <- b ptr
sw.v   vr2, r4      <- a ptr
```

USING VECTOR FLAGS/MASKS TO IMPLEMENT IRREGULAR CONTROL FLOW

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

```
lw.v    vr1, r2
slt.f.vv f1, vr1, vr0
lw.v    vr2, r3,        f1   \
mul.vs  vr3, vr2, r4,   f1    } under mask
sw.v    vr3, r5,        f1   /
```

VECTOR MICROARCHITECTURE



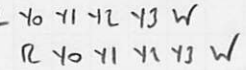
IN C1 AND S1 STAGES WE CAN NOW ALSO REQUEST TO READ OR WRITE A VECTOR OF DATA. SEPARATE LOAD/STORE PIPES WILL EVENTUALLY ALLOW MULTIPLE VECTOR INSTRUCTIONS TO EXECUTE IN PARALLEL.

ASSUME 32 VECTOR REGISTERS, VECTOR REGISTER FILE HAS A TOTAL OF 32 x HWVLEN ELEMENTS

VLMR = VLOAD MEM RESPONSE BUF VSMR = VSTORE MEM REQ BUF

Executing Vector Instructions

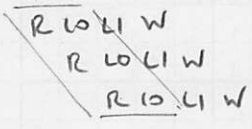
HWVLEN = 2 MUL.VS



HWVLEN = 4 MUL.VS

HWVLEN = 8 MUL.VS

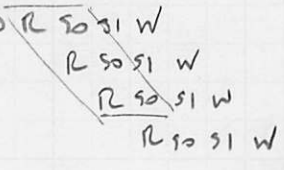
HWVLEN = 4 LW.V



IMPORTANT: WE READ BASE ADDRESS, GENERATE EFFECTIVE ADDRESS, 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 L0 STAGES. WE JUST WRITEBACK THE APPROPRIATE ELEMENT IN THE DESTINATION VECTOR REGISTER

HWVLEN = 4 SW.V



IMPORTANT: WE READ BASE ADDRESS, generate effective ADDRESS, 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

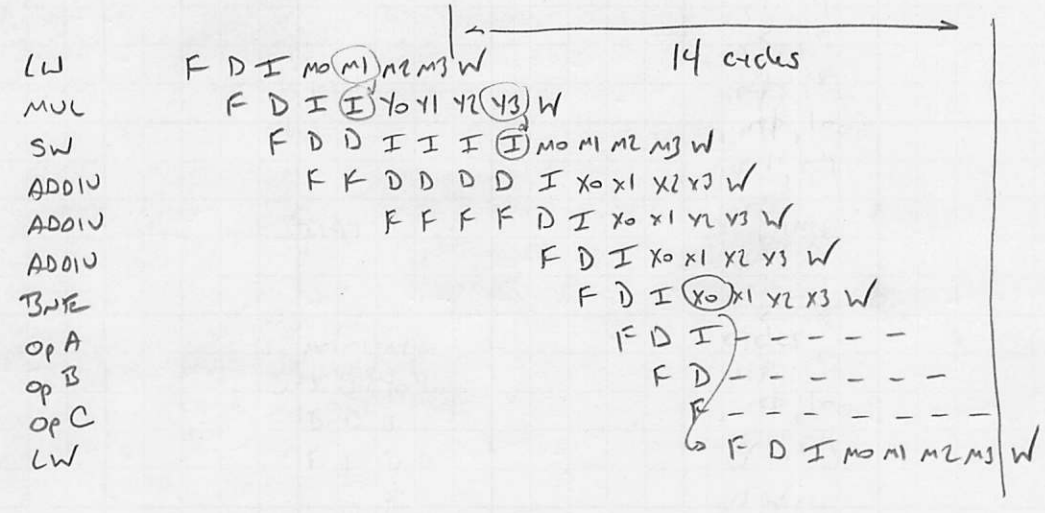
SCALAR ASSEMBLY

```
loop:
  LW    r1, 0(r2)
  MUL   r3, r1, r4
  SW    r3, 0(r5)
  ADDIU r2, r2, 4
  ADDIU r5, r5, 4
  ADDIU r7, r7, -1
  BNE   r7, r0, loop
```

VECTOR ASSEMBLY

```
loop:
  SetVL  r8, r7
  LW.V   vr1, r2
  MUL.VS vr3, vr1, r4
  SW.V   vr3, r5
  SLL    r9, r8, 2
  ADDU   r2, r2, r9
  ADDU   r5, r5, r9
  SUBU   r7, r7, r8
  BNE    r7, r0, loop
```

FOR REFERENCE WHAT IS THE EXECUTION TIME OF SCALAR CODE?



ASSUME 64 ITERATIONS

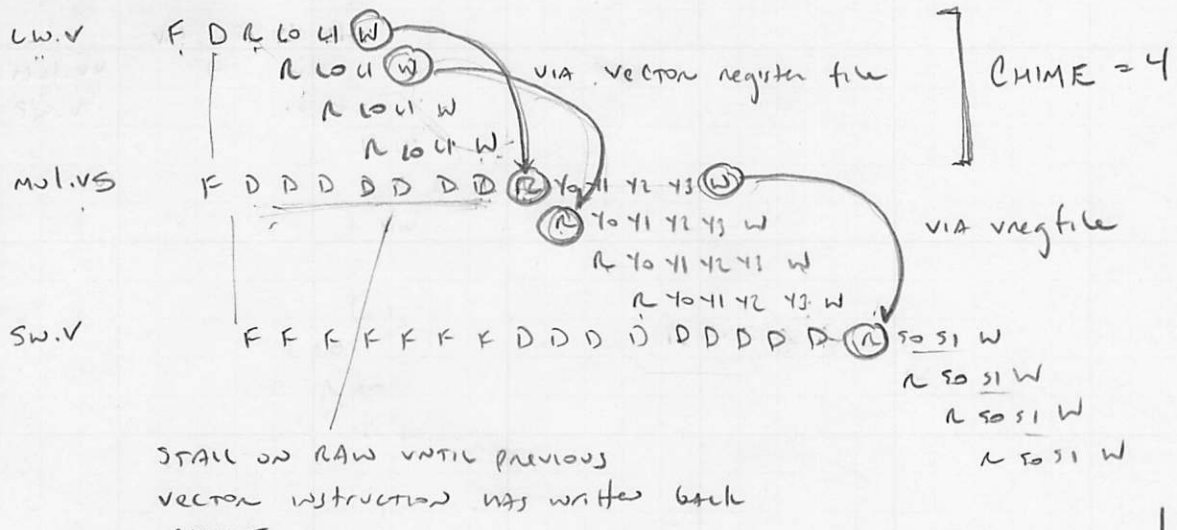
INSTR = 7 x 64 = 448

CYCLES / INSTR = (14 x 64) / 448 = 2

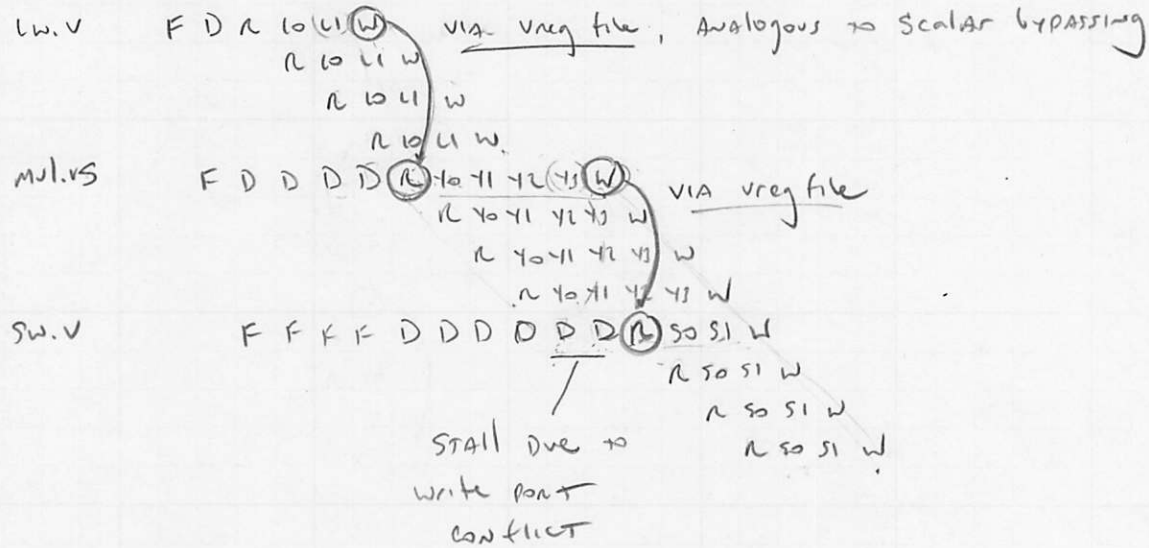
CYCLE TIME = 1ns

TOTAL EXECUTION TIME = 896 ns

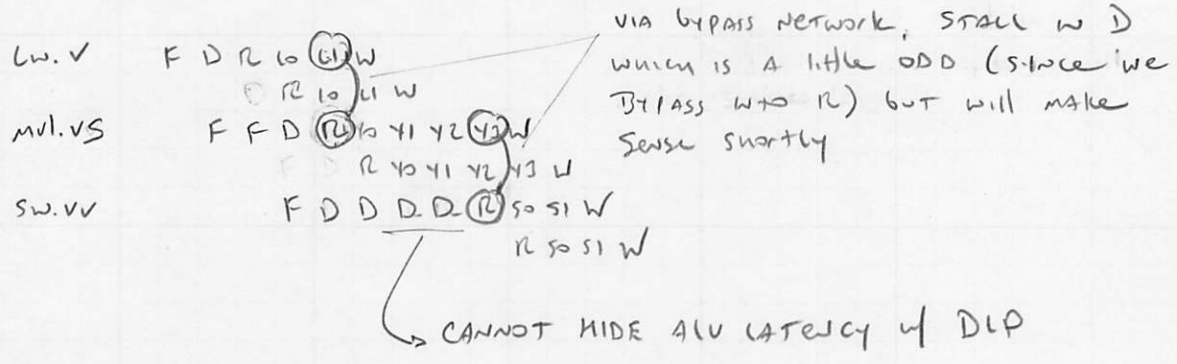
BASIC EXECUTION, HWVLEN = 4



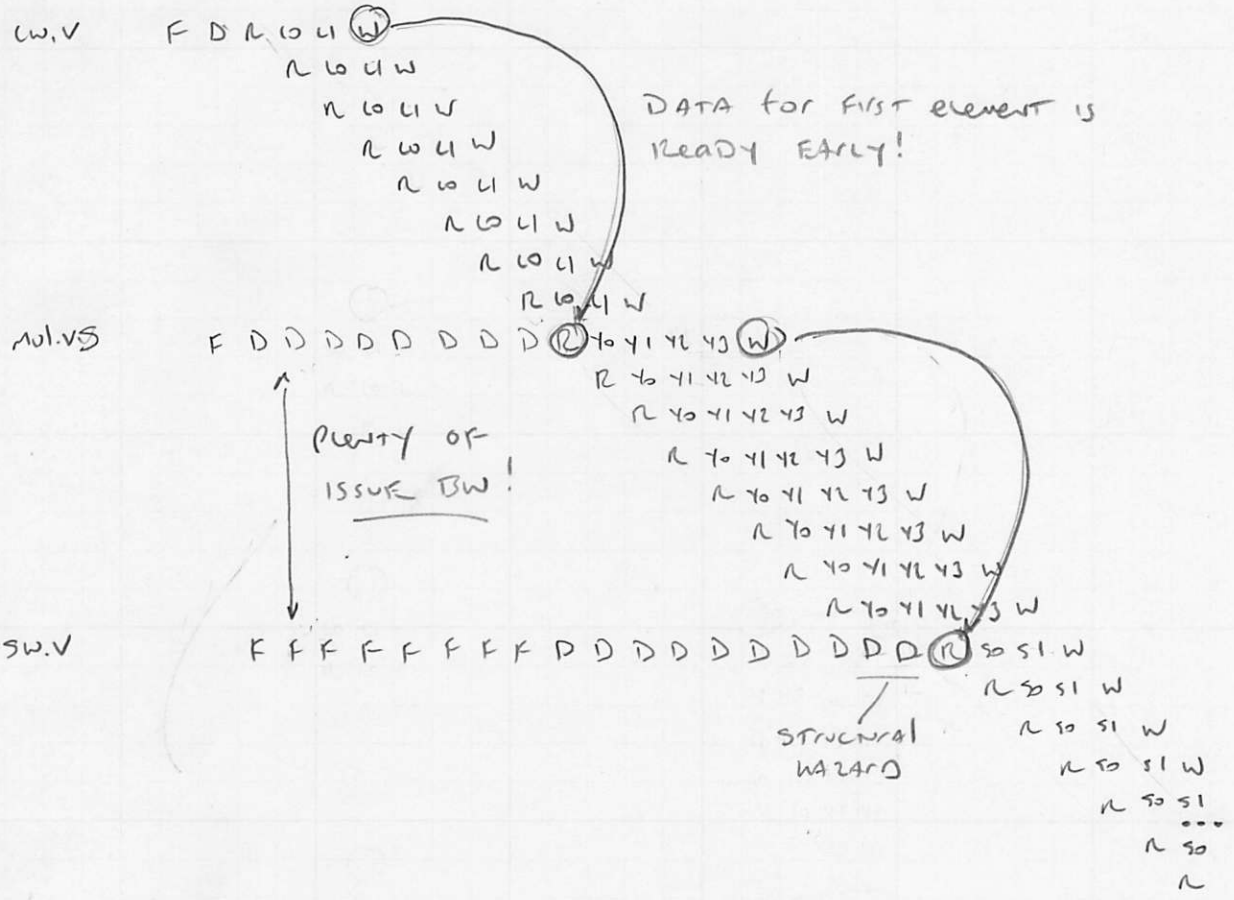
CHAINING, HWVLEN = 4



CHAINING, HWVLEN = 2 (ASSUME R ALSO = 2)



CHAINING, HWVLEN = 8 (ASSUME N = 8)



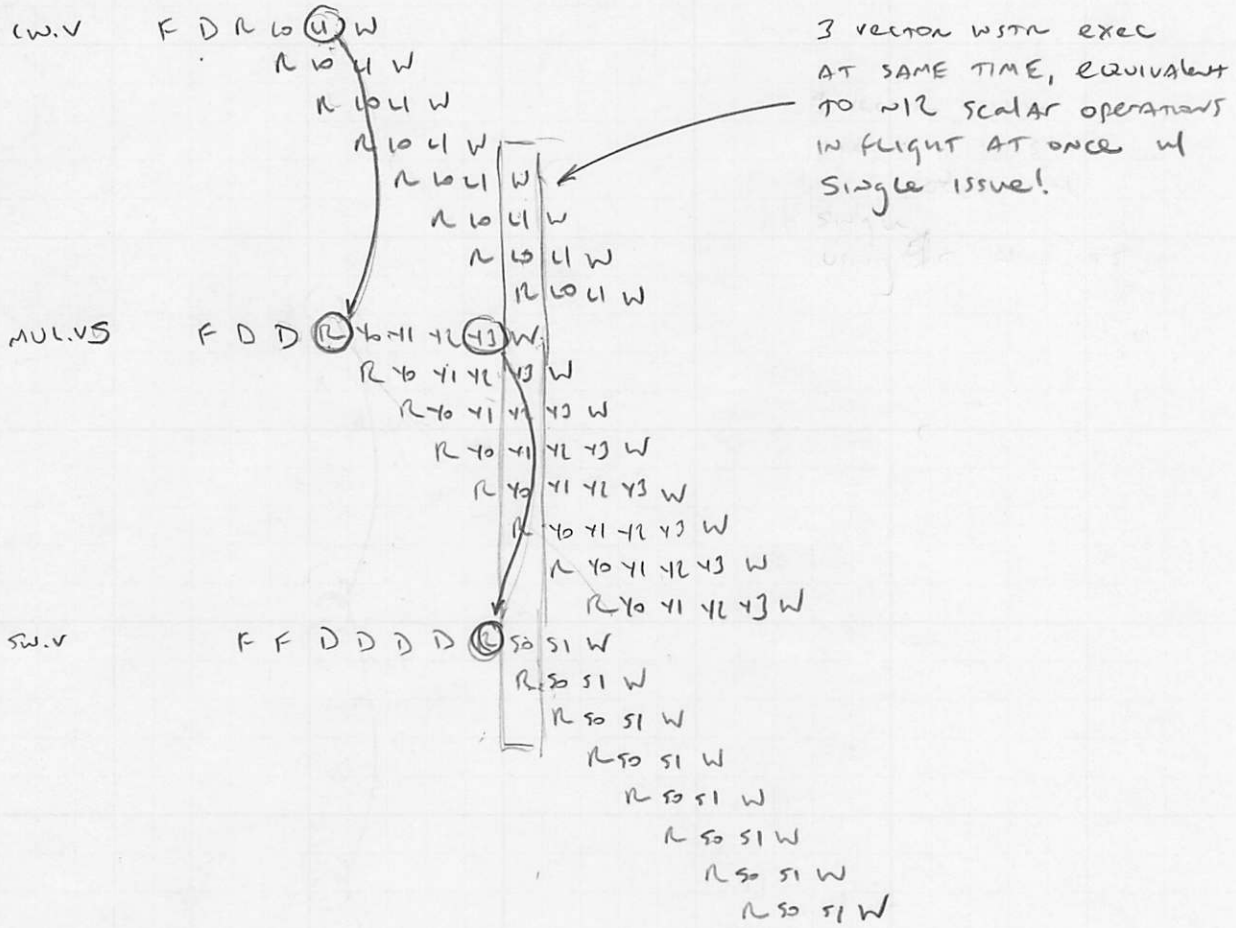
KEY OBSERVATION: WITH RELATIVELY LITTLE FETCH/ISSUE BANDWIDTH + INORDER FETCH/ISSUE, WE CAN STILL KEEP FUNCTIONAL UNITS BUSY FOR MANY CYCLES.

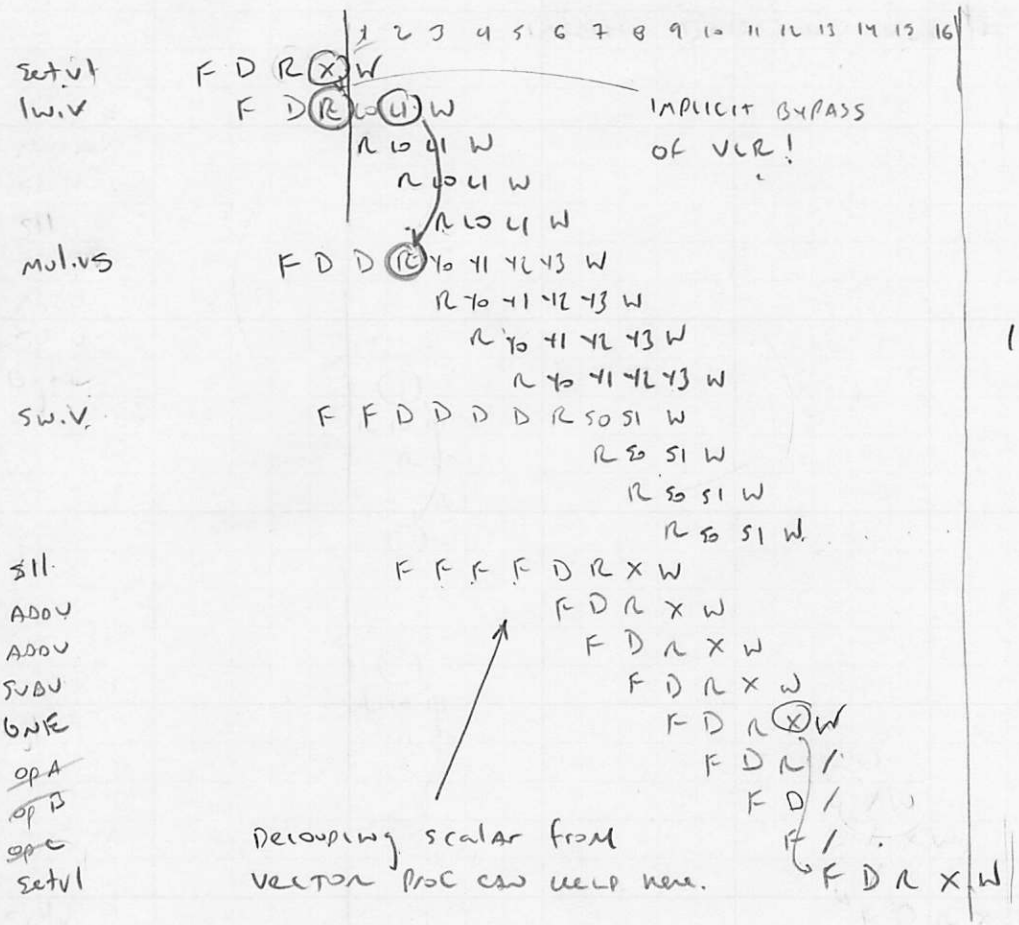
* ADD MORE VREGFILE PORTS TO EXPLOIT VECTOR INSTR. LEVEL PARALLELISM

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	7r	3w

CHAINING, HWVLEN = 8 (ASSUME n=8) w/ 7r 3w VRegFile





INSTR = 9 × 16 = 144

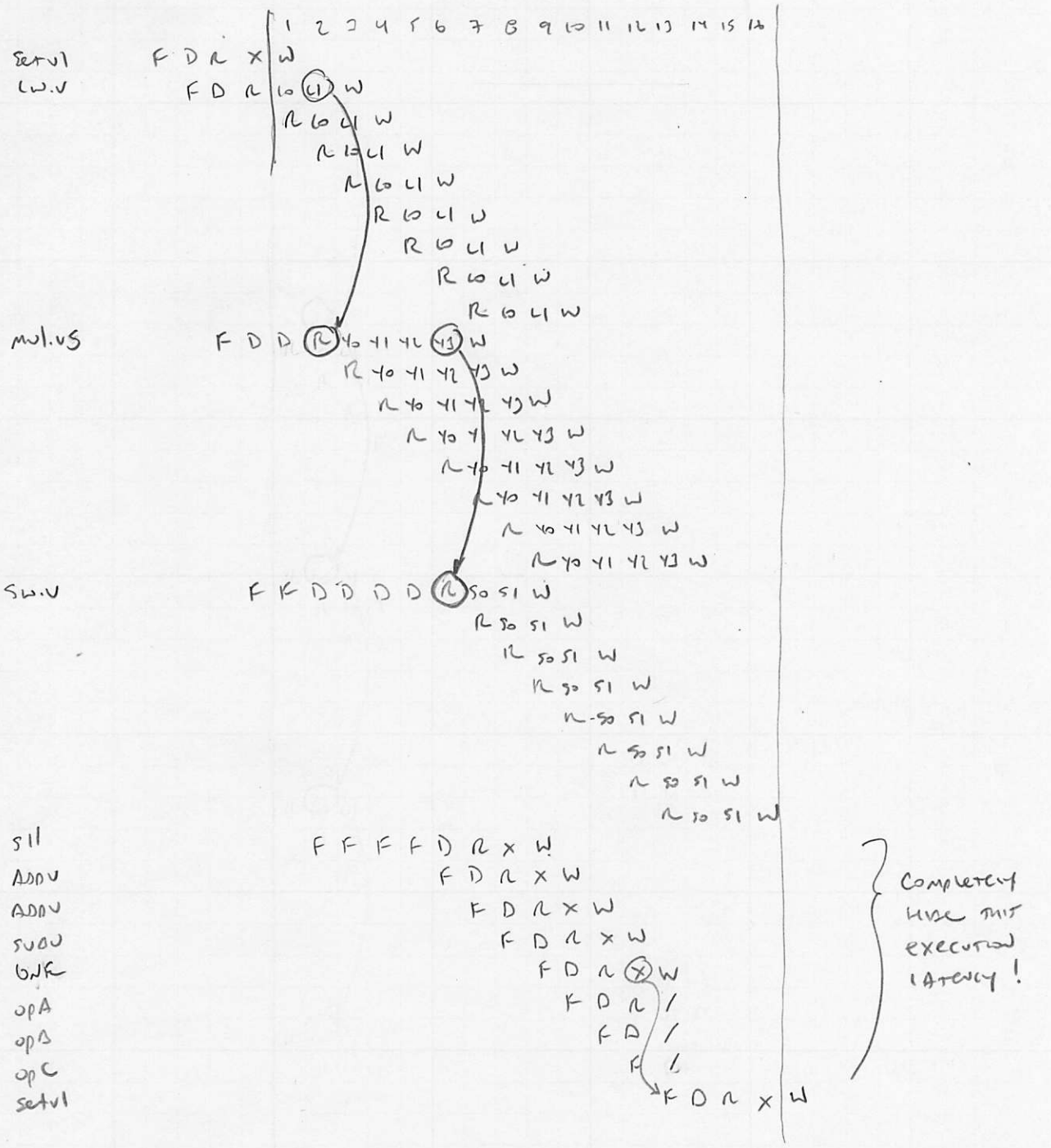
Cycles/INSTR = (16 × 16) / 144 = 1.78

Cycle time = 1 ns

TOTAL EXECUTION TIME = 256 cycles

SPEEDUP OF 3.5 BUT NEEDS TO ADD 4x Registers AND 7r 3w PORTS

WHAT IF MVL = 8?



INSTN = 9 × 8 = 72

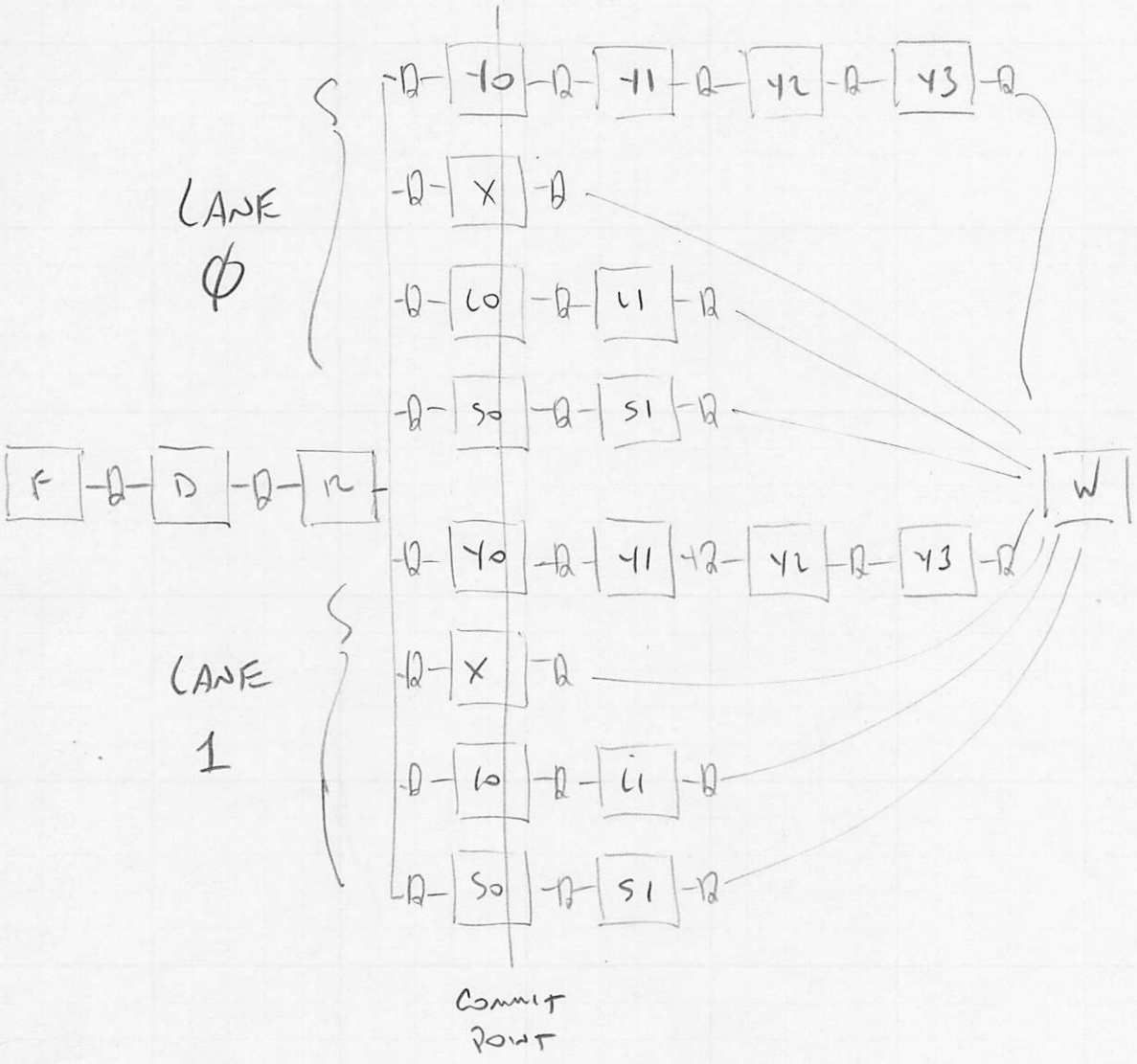
CPI = (16×8)/72 = 1.78

Cycle time = 1ns

TOTAL EXECUTION TIME = 128.16

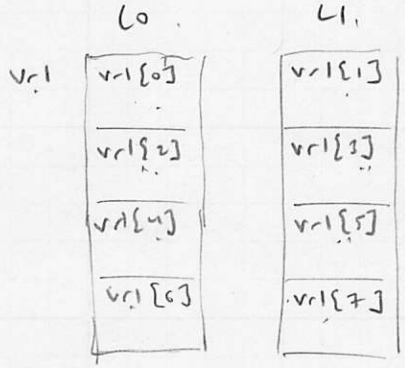
SPEEDUP OF 7× WITH 8× MORE REGISTERS + STILL 7R3W PORTS

MULTIPLE LANES

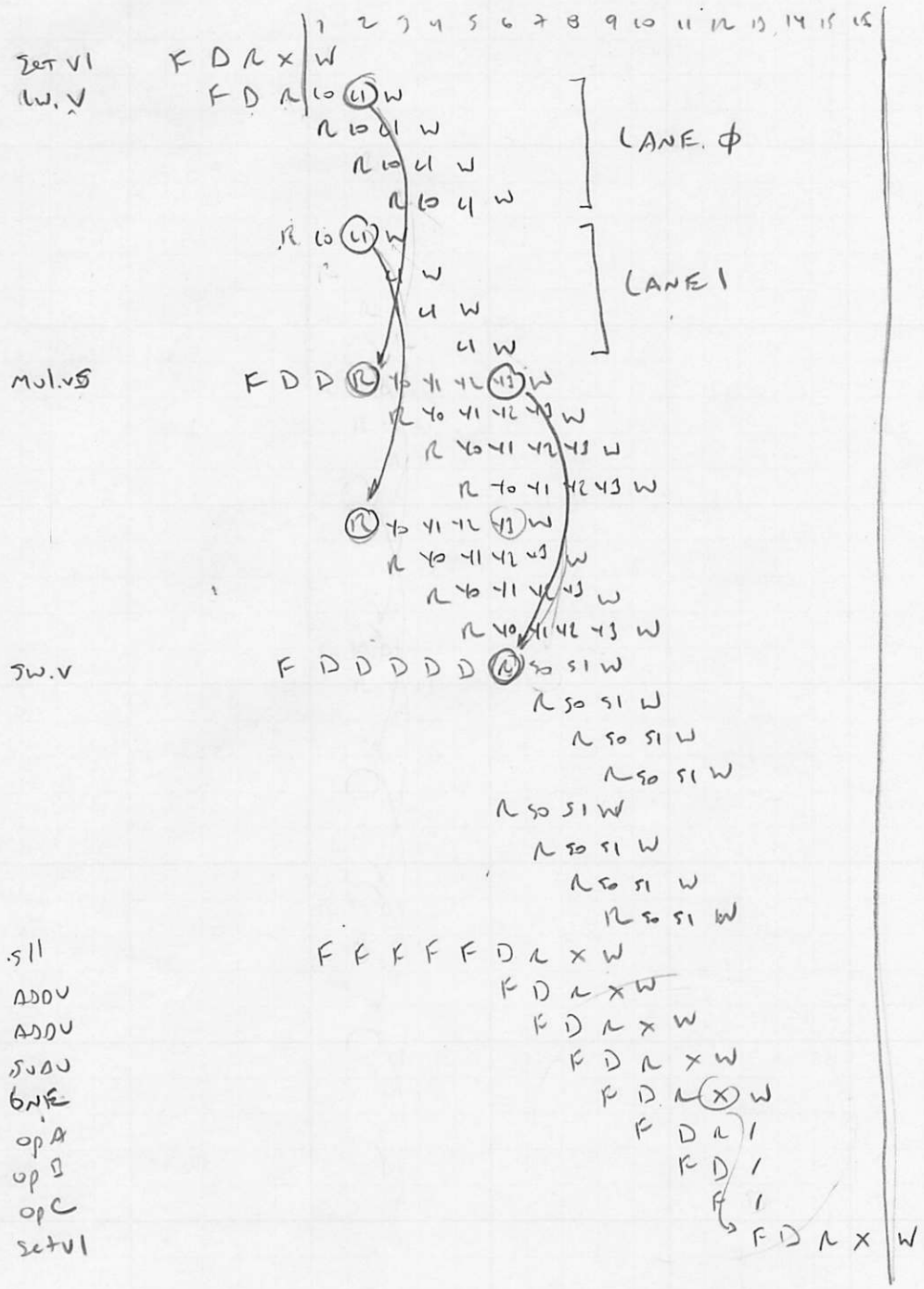


V Register File is partitioned by lane.

Assume HWVLEN = 8 w/ 2 lanes



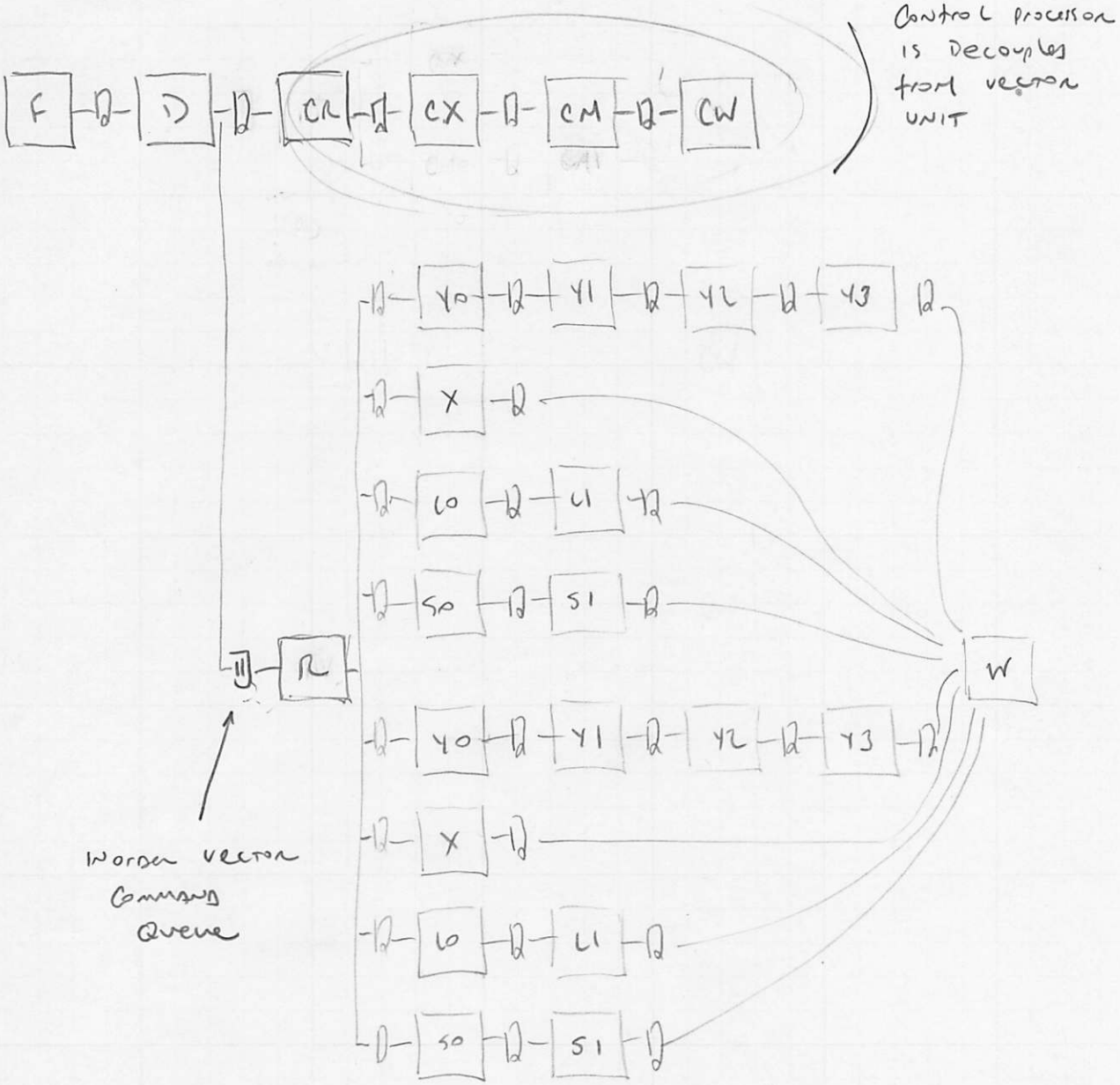
only needs 7r 3w / lane

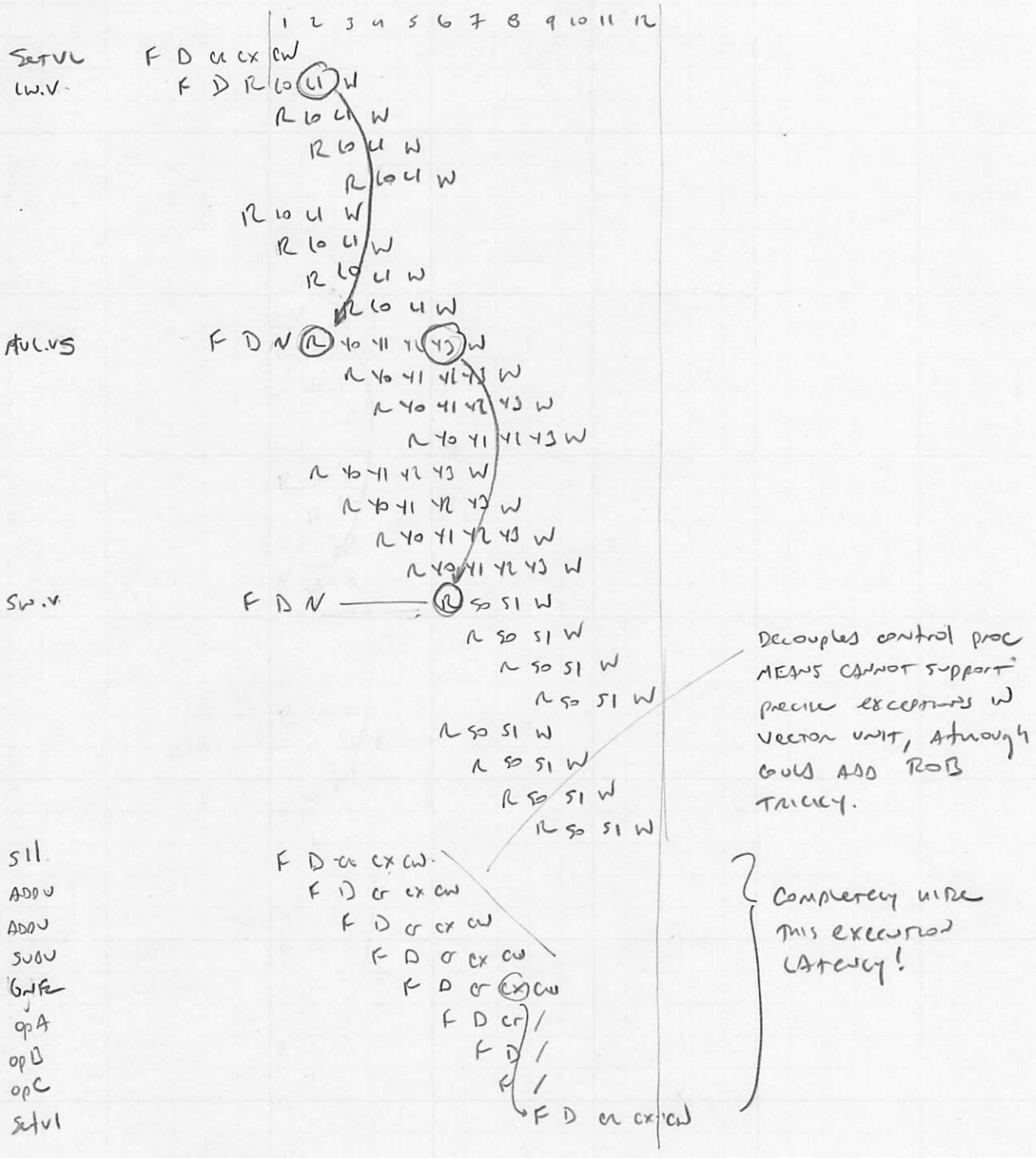


16 cycles / ITA, SAME AS w/ 1 lane 4 vlen = 4

scalar code getting in the way
- ↑ vlen
- Decouple scalar processor

MULTIPLE LANES W/ DECOUPLED CONTROL PROC





INSTR = 9 x 8 = 72

CPI = (12 x 8) / 72 = 1.33

CYCLE TIME = 1 ns

TOTAL EXECUTION TIME = 96 CYCLES!

↳ WITH WORDEN, SINGLE ISSUE!