CSCE 5610: Computer Architecture

Software Pipelining
Consider the code
Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld</td>
<td>f0, 0(x1)</td>
<td></td>
</tr>
<tr>
<td>fadd</td>
<td>f4, f0, f2</td>
<td></td>
</tr>
<tr>
<td>fmul</td>
<td>f6, f4, f4</td>
<td></td>
</tr>
<tr>
<td>fsd</td>
<td>f6, 0(x1)</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>x1, x1, -8</td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>x1, Loop</td>
<td></td>
</tr>
</tbody>
</table>

Consider the dependencies
- fsd needs the results from fmul
- fmul needs results from fadd
- fadd needs the value read by fld

If we have say 2 cycle latency from fld to fadd, 3 from fadd to fmul
and 5 cycle latency from fmul to fsd
how can we avoid these latencies
We will take fld from i+3, fadd from i+2, fmul from i+1 and fsd from i-th iteration
And make these instructions (from different iterations) as a loop body

Loop: fsd  f4, 24(x1)  fsd uses data from fmul from previous iteration
       fmul  f6, f4, f4  there is 5 cycle delay
       fadd  f4, f0, f2  Likewise fmul depends on fadd from previous iteration
       fld   f0, 0(x1)   iteration
       addi x1, x1, -8  fld depends on fld from previous iteration
       bne  x1, Loop

Need preamble that does 3 load, 2 adds, one mult
Need postscript that does one add, 2 mult and 3 stores

Instruction Level Parallelism is limited

What limits the ILP
1. Instruction window width – larger is better
   but more instructions means more complex hardware to detect dependencies
2. Number of Functional units and reservation stations
   how many instructions can be issued
3. Number of reorder buffers
   again limits number of instructions that can be issued
4. Number of renaming registers
   limits on how many WAR and WAWs can be eliminated
5. Branch prediction accuracy
   limits the efficacy of speculative execution
6. Memory address computation – may require indirect computations due to pointers

Multithreading
   SMT \rightarrow interleave instructions from different threads
<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>th1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>th2.1</td>
<td>th1.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>th3.1</td>
<td>th2.1</td>
<td>th1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>th4.1</td>
<td>th3.1</td>
<td>th2.1</td>
<td>th1.1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>th5.1</td>
<td>th4.1</td>
<td>th3.1</td>
<td>th2.1</td>
<td>th1.1</td>
</tr>
<tr>
<td>6</td>
<td>th1.2</td>
<td>th5.1</td>
<td>th4.1</td>
<td>th3.1</td>
<td>th2.1</td>
</tr>
<tr>
<td>7</td>
<td>th2.2</td>
<td>th1.2</td>
<td>th5.1</td>
<td>th4.1</td>
<td>th3.1</td>
</tr>
<tr>
<td>8</td>
<td>th3.2</td>
<td>th2.2</td>
<td>th1.2</td>
<td>th5.1</td>
<td>th4.1</td>
</tr>
<tr>
<td>9</td>
<td>th4.2</td>
<td>th3.2</td>
<td>th2.2</td>
<td>th1.2</td>
<td>th5.1</td>
</tr>
<tr>
<td>10</td>
<td>th5.2</td>
<td>th4.2</td>
<td>th3.2</td>
<td>th2.2</td>
<td>th1.2</td>
</tr>
</tbody>
</table>

Utilization without threads $U_i = R/(R+L)$
CSCE 5610: Computer Architecture

• Thread Level Parallelism
  • What is the maximum Utilization that is possible?
    – Maximum utilization with threads \( U_{\text{max}} = \frac{R}{R+C} \)
      \( C \) is the context switching overhead
  • How many threads are needed to reach \( U_{\text{max}} \)?
    – \( N_{\text{saturation}} = \frac{R+L}{R+C} \)
  • If there are fewer threads?
    – Utilization with \( N \) threads \( U_N = \frac{N^*R}{R+L} \)

Parallel computers
  Multicore
  Vector processors
  GPUs

CSCE 5610: Computer Architecture

So, if we use threads from the same process, we may lose performance due to synchronization

Chapter 4 and 5 describe how to obtain higher levels of performance but using two different types of architectures.
  Chapter 4 looks at Vector processors (including GPUs)
  Chapter 5 looks at multicore systems (including multiple threads on each core)

But before that let us think of the different possibilities for parallel architectures

Flynn’s Classification

SISD
SIMD or SPMD
MISD
MIMD
Consider a program segment like
\[
\text{for } (i = 1; i < n; i++) \{ A(i) = B(i) \text{ op } C(i) \}
\]
For SIMD, loop index \( i \) becomes an AU number.

Data parallelism vs Function parallelism

Parallel programming languages -- most have origins in FORTRAN
loop level parallelism = data parallelism

```plaintext
#pragma omp parallel for
for (i=0; i<n; i++)
{   loop body}
```
We can execute each iteration in parallel (if there are no dependencies among iterations)

CSCE 5610: Computer Architecture

MIMD organizations come in two forms
Shared memory vs Distributed memory.

Programming MIMD systems
Shared memory: threads sharing common data
Message passing: Send and receive data from other processors
Array Processors versus Vector Processors
   Array processors execute the same instruction on different data elements
   Vector processors have special vector instructions
       may or may not have multiple functional units
   Better use of pipelined arithmetic unit

Vector instructions
   Vector ADD

What are the operands? CDC Star 100
   VADD Ra, Ri, Rb, Rj, Re, Rk, Rn
   Need to initialize these registers

Cray Vector Unit
   Vector Registers (64 words per register)
   VADD Vi, Vj, Vk
   Additional registers for specifying vector length and mask
   Need to load vector elements into these vector registers
CSCE 5610: Computer Architecture

Single Port Memory
- 16 banks of 64-bit words
- 8-bit SECDED
- 80MW/sec data load/store
- 320MW/sec instruction buffer refill

- Memory bank cycle 50 ns
- Processor cycle 12.5 ns (80MHz)

Vector Code Example

# C code
for (i=0; i<64; i++)
C[i] = A[i] + B[i];

# Scalar Code
LI R4, 64
loop:
L.D F0, 0(R1)
L.D F2, 0(R2)
ADD.D F4, F2, F0
S.D F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ R4, loop

# Vector Code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3
In Chapter 4 there is a vector extension to RISC V. We will use this for our examples. Vector instructions are shown on page 286.

### Vector Instructions (RISC-V)

#### Register Definitions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Sign-extend of V0[i] and Y0[i], then put each result in V0[i]</td>
</tr>
<tr>
<td>v1</td>
<td>Sign-extend of V1[i] and Y1[i], then put each result in V1[i]</td>
</tr>
<tr>
<td>v2</td>
<td>Sign-extend of V2[i] and Y2[i], then put each result in V2[i]</td>
</tr>
<tr>
<td>v3</td>
<td>Sign-extend of V3[i] and Y3[i], then put each result in V3[i]</td>
</tr>
<tr>
<td>v4</td>
<td>Logical AND of V0[i] and Y0[i], then put each result in V0[i]</td>
</tr>
<tr>
<td>v5</td>
<td>Logical AND of V1[i] and Y1[i], then put each result in V1[i]</td>
</tr>
<tr>
<td>v6</td>
<td>Logical AND of V2[i] and Y2[i], then put each result in V2[i]</td>
</tr>
<tr>
<td>v7</td>
<td>Logical AND of V3[i] and Y3[i], then put each result in V3[i]</td>
</tr>
</tbody>
</table>

#### Instructions

1. **vsetdcfg 4*FP64**
   - Enable four 64-bit floating-point vector registers.

2. **vsetdcfg 4*FP32**
   - Enable four 32-bit floating-point vector registers.

3. **vsetdcfg 4*FP16**
   - Enable four 16-bit floating-point vector registers.

4. **vsetdcfg 4*INT64**
   - Enable four 64-bit integer vector registers.

5. **vsetdcfg 4*INT32**
   - Enable four 32-bit integer vector registers.

6. **vsetdcfg 4*INT16**
   - Enable four 16-bit integer vector registers.

7. **vsetdcfg 4*INT8**
   - Enable four 8-bit integer vector registers.

8. **vsetdcfg 4*FP128**
   - Enable four 128-bit floating-point vector registers.

9. **vsetdcfg 4*FP64**
   - Enable four 64-bit floating-point vector registers.

10. **vsetdcfg 4*FP32**
    - Enable four 32-bit floating-point vector registers.

11. **vsetdcfg 4*FP16**
    - Enable four 16-bit floating-point vector registers.

12. **vsetdcfg 4*INT64**
    - Enable four 64-bit integer vector registers.

13. **vsetdcfg 4*INT32**
    - Enable four 32-bit integer vector registers.

14. **vsetdcfg 4*INT16**
    - Enable four 16-bit integer vector registers.

15. **vsetdcfg 4*INT8**
    - Enable four 8-bit integer vector registers.

16. **vsetdcfg 4*FP128**
    - Enable four 128-bit floating-point vector registers.

17. **vsetdcfg 4*FP64**
    - Enable four 64-bit floating-point vector registers.

18. **vsetdcfg 4*FP32**
    - Enable four 32-bit floating-point vector registers.

19. **vsetdcfg 4*FP16**
    - Enable four 16-bit floating-point vector registers.

20. **vsetdcfg 4*FP128**
    - Enable four 128-bit floating-point vector registers.

21. **vsetdcfg 4*FP64**
    - Enable four 64-bit floating-point vector registers.

22. **vsetdcfg 4*FP32**
    - Enable four 32-bit floating-point vector registers.

23. **vsetdcfg 4*FP16**
    - Enable four 16-bit floating-point vector registers.

24. **vsetdcfg 4*FP128**
    - Enable four 128-bit floating-point vector registers.

25. **vsetdcfg 4*FP64**
    - Enable four 64-bit floating-point vector registers.

26. **vsetdcfg 4*FP32**
    - Enable four 32-bit floating-point vector registers.

27. **vsetdcfg 4*FP16**
    - Enable four 16-bit floating-point vector registers.

28. **vsetdcfg 4*FP128**
    - Enable four 128-bit floating-point vector registers.

29. **vsetdcfg 4*FP64**
    - Enable four 64-bit floating-point vector registers.

30. **vsetdcfg 4*FP32**
    - Enable four 32-bit floating-point vector registers.

31. **vsetdcfg 4*FP16**
    - Enable four 16-bit floating-point vector registers.

32. **vsetdcfg 4*FP128**
    - Enable four 128-bit floating-point vector registers.

33. **vsetdcfg 4*FP64**
    - Enable four 64-bit floating-point vector registers.

#### Examples

- **Example 1:**

  **V = a * X + Y**

  In C:

  ```c
  for (i=0; i<n; i++) { Y[i] = a*X[i] + Y[i]; }
  ```

- **Example 2:**

  **Show the code for RV64G and RV64V for the DAXPY loop.**

  **Answer:**

  **RV64G Code:**

  ```riscv
  fld      f0,a              # Load scalar a
  addi     x28,x5,#256       # Last address to load
  Loop: fld      f1,0(x5)          # Load X[i]
  fmul.d   f1,f1,f0          # a * X[i]
  fld      f2,0(x6)          # Load Y[i]
  fadd.d   f2,f2,f1          # a * X[i] + Y[i]
  fsd      f2,0(x6)          # Store into Y[i]
  addi     x5,x5,#8          # Increment index to X
  addi     x6,x6,#8          # Increment index to Y
  bne      x28,x5,Loop       # Check if done
  ```

  **RV64V Code:**

  ```riscv
  vsetdcfg   4*FP64          # Enable 4 DP FP vregs
  fld        f0,a            # Load scalar a
  vld        v0,x5           # Load vector X
  vmul       v1,v0,f0        # Vector-scalar mult
  vld        v2,x6           # Load vector Y
  vadd       v3,v1,v2        # Vector-vector add
  vst        v3,x6           # Store the sum
  vdisable                   # Disable vector regs
  ```

  Note: The assembler determines which version of the vector operations to generate. Because the multiply has a scalar operand, it generates `vmul.vs`, whereas the add doesn’t, so it generates `vadd.vv`.

  The most dramatic difference between the scalar and vector code is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 8 instructions versus 258 for RV64G. This reduction occurs because the vector operations work on 32 elements and the overhead instructions that constitute nearly half the loop on RV64G are not present in the RV64V code. When the compiler produces vector instructions for such a sequence, and the resulting code spends much of its time running in vector mode, the code is said to be vectorized or vectorizable. Loops can be vectorized when they do not have dependences between iterations of a loop, which are called loop-carried dependences (see Section 4.5).

  Another important difference between RV64G and RV64V is the frequency of pipeline interlocks for a simple implementation of RV64G. In the straightforward RV64G code, every `fadd.d` must wait for a `fmul.d`, and every `fsd` must wait for the `fadd.d`. On the vector processor, each vector instruction will stall only for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus pipeline stalls are required only once per vector instruction, rather than once per vector element. Vector architects call forwarding of element-dependent operations chaining, in that the dependent operations are "chained" together. In this example, the pipeline stall frequency on RV64G will be about 32 × higher than it is on RV64V. Software pipelining, loop unrolling (Appendix H), or out-of-order execution can reduce the pipeline stalls on RV64G; however, the large difference in instruction bandwidth cannot be reduced substantially.

#### Dynamic Register Typing

We assume that each Vector register contains 32 registers, each 64 bits long.
Example

Show the code for RV64G and RV64V for the DAXPY loop. For this example, assume that X and Y have 32 elements and the starting addresses of \( X \) and \( Y \) are in \( x5 \) and \( x6 \), respectively. (A subsequent example covers when they do not have 32 elements.)

Answer

Here is the RISC-V code:

\[
\begin{align*}
\text{fld} & \quad f0,a & \quad \text{# Load scalar a} \\
\text{addi} & \quad x28,x5,#256 & \quad \text{# Last address to load} \\
\text{Loop:} & \quad \text{fld} & \quad 0(x5) & \quad \text{# Load } X[i] \\
& \quad \text{fmul.d} & \quad f1,f1,f0 & \quad \text{# } a \times X[i] \\
& \quad \text{fld} & \quad 0(x6) & \quad \text{# Load } Y[i] \\
& \quad \text{fadd.d} & \quad f2,f2,f1 & \quad \text{# } a \times X[i] + Y[i] \\
& \quad \text{fsd} & \quad 0(x6) & \quad \text{# Store the sum} \\
& \quad \text{addi} & \quad x5,x5,#8 & \quad \text{# Increment index to } X \\
& \quad \text{addi} & \quad x6,x6,#8 & \quad \text{# Increment index to } Y \\
& \quad \text{bne} & \quad x28,x5,Loop & \quad \text{# Check if done}
\end{align*}
\]

Here is the RV64V code for DAXPY:

\[
\begin{align*}
\text{vsetdcfg} & \quad 4*\text{FP64} & \quad \text{# Enable 4 DP FP vregs} \\
\text{fld} & \quad f0,a & \quad \text{# Load scalar a} \\
\text{vld} & \quad v0,x5 & \quad \text{# Load vector } X \\
\text{vmul} & \quad v1,v0,f0 & \quad \text{# Vector-scalar mult} \\
\text{vld} & \quad v2,x6 & \quad \text{# Load vector } Y \\
\text{vadd} & \quad v3,v1,v2 & \quad \text{# Vector-vector add} \\
\text{vst} & \quad v3,x6 & \quad \text{# Store the sum} \\
\text{vdisable} & \quad & \quad \text{# Disable vector regs}
\end{align*}
\]

Note that the assembler determines which version of the vector operations to generate. Because the multiply has a scalar operand, it generates \( \text{vmul.vs} \), whereas the add doesn’t, so it generates \( \text{vadd.vv} \).

The initial instruction configures the first four vector registers to hold 64-bit floating-point data. The last instruction disables all vector registers. If a context switch happened after the last instruction, there is no additional state to save.

The most dramatic difference between the preceding scalar and vector code is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 8 instructions versus 258 for RV64G. This reduction occurs because the vector operations work on 32 elements and the overhead instructions that constitute nearly half the loop on RV64G are not present in the RV64V code. When the compiler produces vector instructions for such a sequence, and the resulting code spends much of its time running in vector mode, the code is said to be \textit{vectorized} or \textit{vectorizable}. Loops can be vectorized when they do not have dependences between iterations of a loop, which are called loop-carried dependences (see Section 4.5).

Another important difference between RV64G and RV64V is the frequency of pipeline interlocks for a simple implementation of RV64G. In the straightforward RV64G code, every \( \text{fadd.d} \) must wait for a \( \text{fmul.d} \), and every \( \text{fsd} \) must wait for the \( \text{fadd.d} \). On the vector processor, each vector instruction will stall only for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus pipeline stalls are required only once per vector instruction, rather than once per vector element. Vector architects call forwarding of element-dependent operations \textit{chaining}, in that the dependent operations are “chained” together. In this example, the pipeline stall frequency on RV64G will be about 32 times higher than it is on RV64V. Software pipelining, loop unrolling (Appendix H), or out-of-order execution can reduce the pipeline stalls on RV64G; however, the large difference in instruction bandwidth cannot be reduced substantially.

Let’s show off the dynamic register typing before discussing performance of the code.
Consider the example we have seen already (and ignore the scalar load)

\begin{verbatim}
  vsetdcfg 4*FP64
  fld  f0, a : load scalar a
  vld  v0, x5 : load vector X into v0
  vmul v1, v0, f0 : compute a*X
  vld  v2, x6 : load vector Y into v2
  vadd v3, v1, v2 : compute a*X + Y
  vsd v3, x6 : store Y
  vdisable
\end{verbatim}

We have 3 convoys (ignoring scalar load)

1. vld v0, x5 vmul v1, v0, f0
2. vld v2, x6 vadd v3, v1, v2
3. vsd v3, x6

Since we have only one Load/Store vector unit, we have structural hazards.

The code takes 3 chimes.

How many vector operations did we complete (in total) = 5
How many vector floating point operations = 2

Or we achieved 1.5 chimes per vector FLOP.

The number of actual operations we can complete depends on the number of vector elements in a vector register.

In our example we have 32 vector elements. So we completed 32 operations in 1.5 chimes
One more example and the concept of reduction

Consider a simple vector dot product \( X^* Y \)

\[
\begin{align*}
\text{sum} &= 0.0; \\
\text{for (i=0; i<n; i++) sum} &= \text{sum + X}[i]^* Y[i]; \\
\end{align*}
\]

- \text{vsetdcfg} 3*FP64 ; reserve 3 vector registers
- \text{vld} v1, x1 ; load X in v1
- \text{vld} v2, x2 ; load Y in v2
- \text{vmul} v3, v1, v2 ; vector multiply
- \text{vredplus} f0, v3 ; f0 is sum of elements of v3
- \text{fsd} f0, 0(x3) ; store result

Reduction is a very important operation in vector arithmetic

Consider now how we extend this for Matrix Vector product
\[ Y = A^* X \] (A is a matrix, X and Y are vectors or one dimensional arrays)

How to improve performance of vector processors?

a). Can we use multiple functional units to improve the performance of a vector operation?
   for example multiple vector LD/ST units  
   multiple floating point ADD and MULT units

b). What if the array size is not a multiple of 32?

c). How to handle if conditions that require vector operations on some array elements only?
   may be only certain elements of the vector are involved
   for (i=0; i<n; i++)
   if (A[i] <> 0) C[i] = B[i]/A[i];

d). How to supply the memory bandwidth to load vector registers?

e). What if the array elements are not adjacent? Or consider the following example
   for (i=0; i<n; i++)
   \[ X[a[i]] = X[a[i]] \times Y[b[i]]; \]
CSCE 5610: Computer Architecture

a). Can we use multiple functional units to improve the performance of a vector operation?
Yes – text calls this multiple lanes -- each functional unit operates on a set of vector elements

See page 293-295

CSCE 5610: Computer Architecture

B has 4 vector adders
You feed 4 pairs of inputs
(we feed array elements cyclically)
b). What if the array size is not a multiple of 32?

Most vector processors have a special register called vector length register (vl)
Use a vector length register to indicate the number of elements less than 32
So if you have 132 elements, we first operate on 32 elements 4 times and then
set vl to 4

There is also another register called mvl which defines maximum size
in our example mvl = 32

What if the length is not statically known?

Consider for example
for (i=0; i<n; i++)
    Y[i] = a*X[i] + Y[i];

```
fld f0.n  # Load scalar n
loop:    setvl t0,a0   # vl = t0 = min(mvl,n)
         vld v0,x5     # Load vector X
         slli t1,t0,3   # t1 = vl * 8 (in bytes) – shift left 3 times
         add x5,x5,t1   # Increment pointer to X by vl+8
         vmul v0,v0,f0  # Vector-scalar mult
         vld v1,x6     # Load vector Y
         vadd v1,v0,v1   # Vector-vector add
         sub a0,a0,t0   # n -= vl (t0)
         vst v1,x6     # Store the sum into Y
         add x6,x6,t1   # Increment pointer to Y by vl * 8
         bnez a0,loop   # Repeat if n != 0
```
Predicate registers: If we want to select some elements of an array
For example consider (page 297)
for (i=0; i<32; i++)
    if (X[i] != 0) X[i] – Y[i];

We use predicate register: For our case, we will assume a 32 bit register
Each bit is associated with one element of the vector
We set these bits based on the condition

vsetdefg 2*FP64  # Enable 2 64b FP vector regs
vsetpfgi 1  # Enable 1 predicate register
vld v0,x5  # Load vector X into v0
vld v1,x6  # Load vector Y into v1
fmv.d.x f0,x0  # Put (FP) zero into f0
vpne p0, v0, fo  # set predicate register p0 (p0[i] = 1 if v0[i] != 0)
vsub v0,v0,v1  # Subtract under vector mask
vst v0,x5  # Store the result in X
vdisable  # Disable vector registers
vpdisable  # Disable predicate registers