Review: Software pipelining.

Let us consider the following code:

```
L1:   LD  F0, 0(R1)
      MULD F0, F0, F2
      ADDD F0, F0, F4
      SD   0(R1), F0
      DADDI R1, R1, #8
      DSUB R2, R1, R3
      BNEZ R2, L1
```

Assume that LD/SD have 2 cycle latency, MULD has 5 cycle latency and ADDD has 3 cycle latency. Integer instructions (DADDI, DSUB, BNEZ) have one cycle latency.

Let us first use 2 issue speculative execution to track these instructions:

<table>
<thead>
<tr>
<th>Itr</th>
<th>Instruction</th>
<th>issue</th>
<th>execute</th>
<th>memory</th>
<th>CDB</th>
<th>Commit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Wait for LD</td>
</tr>
<tr>
<td>1</td>
<td>MULD F0, F0, F2</td>
<td>1</td>
<td>5–9</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Wait for LD</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F0, F0, F4</td>
<td>2</td>
<td>11–13</td>
<td>14</td>
<td>15</td>
<td></td>
<td>Wait for MULD</td>
</tr>
<tr>
<td>1</td>
<td>SD 0(R1), F0</td>
<td>2</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>Wait for ADDD</td>
</tr>
<tr>
<td>1</td>
<td>DADDI R1, R1, #8</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DSUB R2, R1, R3</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BNEZ R2, L1</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>21</td>
<td></td>
<td>Wait for DSUB (R2)</td>
</tr>
<tr>
<td>2</td>
<td>LD F0, 0(R1)</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>22</td>
<td>Wait for DADDI (R1) to complete</td>
</tr>
<tr>
<td>2</td>
<td>MULD F0, F0, F2</td>
<td>5</td>
<td>9–14</td>
<td>15</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDD F0, F0, F4</td>
<td>5</td>
<td>16–18</td>
<td>19</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1), F0</td>
<td>6</td>
<td>20</td>
<td>21</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DADDI R1, R1, #8</td>
<td>6</td>
<td>7</td>
<td>11</td>
<td>26</td>
<td></td>
<td>several CDB conflicts</td>
</tr>
<tr>
<td>2</td>
<td>DSUB R2, R1, R3</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BNEZ R2, L1</td>
<td>7</td>
<td>13</td>
<td>15</td>
<td>28</td>
<td></td>
<td>CDB conflict with ADDD</td>
</tr>
</tbody>
</table>

NOTE: the highest numbered cycle here is 21 when second SD completes.

Now, let us use software pipelining.
For the preamble we need 3 LDs, 2 MULD and 1 ADDD completed

Preamble:
LD F6, 0(R1)
MULD F6, F6, F2
ADDD F0, F6, F4
LD F8, 0(R1)
MULD F8, F8, F2
LD F10, 16(R1)
ADDD R1, R1, #24
DSUB R2, R1, R3
BZ R2, postscript

The software pipelined loop is entered after preamble

L1:
SD -24(R1), F0
ADDD F0, F8, F4
MULD F8, F10, F2
LD F10, 0(R1)
ADDD R1, R1, #8
DSUB R2, R1, R3
BNEZ R2, L1

Note the values of offsets in LD/SD instructions and also how R1 is incremented. Finally note that if R1 already exceed the loop limit, we need to exit. Note how we renamed registers so that the values in F0 are not destroyed.

We also need complete some computations outside of the loop

Postscript: SD -24(R1), F0
ADDD F0, F8, F4
MULD F8, F10, F2
ADDD F0, F8, F4
SD -16(R1), F0

Let track the instructions inside the new loop using 2-issue speculative execution

Note how we renamed registers so that the values in F0 are not destroyed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>issue</th>
<th>execute</th>
<th>memory</th>
<th>CDB</th>
<th>Commit</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD -24(R1), F0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F0, F8, F4</td>
<td>1</td>
<td>2–4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULD F8, F10, F2</td>
<td>2</td>
<td>3–7</td>
<td>8</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F10, 0(R1)</td>
<td>2</td>
<td>3–7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>DADDI R1, R1, #8</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSUB R2, R1, R3</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>CDB conflict</td>
<td></td>
</tr>
<tr>
<td>BNEZ R2, L1</td>
<td>4</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD -24(R1), F0</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>13</td>
<td>Wait for ADD from previous iteration</td>
<td></td>
</tr>
<tr>
<td>ADDD F0, F8, F4</td>
<td>5</td>
<td>9–11</td>
<td>12</td>
<td>14</td>
<td>Wait for MULD from previous iteration</td>
<td></td>
</tr>
<tr>
<td>MULD F8, F10, F2</td>
<td>5</td>
<td>7–11</td>
<td>13</td>
<td>15</td>
<td>Wait for LD from previous iteration</td>
<td></td>
</tr>
<tr>
<td>LD F10, 0(R1)</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>16</td>
<td>Wait for R1 to be incremented</td>
</tr>
<tr>
<td>DADDI R1, R1, #9</td>
<td>6</td>
<td>7–11</td>
<td>14</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSUB R2, R1, R3</td>
<td>7</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEZ R2, L1</td>
<td>7</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Now the highest numbered cycles in 18 when second BNEZ places its result on CDB
CSCE 4610: Computer Architecture

Read the section of Chapter 3 about how much instruction level parallelism we can really get from applications.

We cannot get a large amount of ILP using out of order execution because of limited resources:
- Instruction window size
- Reservation stations and functional units
- Reorder buffers
- Renaming registers
- Accuracy of branch prediction
- Address computation (for indirect addresses, pointer dereferencing)
- Data dependencies

We may want to explore multithreading to gain further performance improvements.
At least we can minimize data dependencies since instructions from different threads are unlikely to have data dependencies.

Hardware support for multiple threads?
- Multiple register sets
- Multiple PCs

CSCE 4610 April 13, 2017
Thread Level Parallelism – computing performance

- Consider a single CPU and one thread
  - When a thread issues a "long latency" operation, the thread is blocked and CPU idles

- Let $L$ be amount of time needed for the long latency operation, and let $R$ be the average amount of time between such latency operations

Utilization without threads $U_1 = \frac{R}{R+L}$

Thread Level Parallelism

- Consider a single CPU supporting multiple threads
  - When a thread issues a "long latency" operation, the thread is context switched and a new thread starts execution.

- In addition to $R$ and $L$, let us assume that $C$ is the time needed to context switch
CSCE 4610: 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 $U_N = \frac{N \cdot R}{R+L}$

If we assume a context switch on every cache miss, $R$ can be equated to $\frac{1}{\text{cache miss rate}}$ and $L$ as the miss penalty.

Issues to deal with concerning multi-threading

If threads belong to a single process, and share data
- need to coordinate modifications to shared data (mutual exclusion)
- need to synchronize thread execution (barriers)
CSCE 4610: Computer Architecture

Let us see why we need synchronization among multiple threads in the same process.

```c
void compute_min(j)
{
    local_min = min();
    lock(lock_variable);
    if (local_min < global_min)
        global_min = local_min;
    unlock(lock_variable);
    exit();
}

for (j=0; j<n; j++)
    thread_id[j] = spawn_thread(compute_min, j);
for (j=0; j<n; j++)
    join(thread_id[j]);
```

Let us see how to parallelize this.

Why do we need locks?

b). for (j=0; j<n; j++)
   if (local_min(j) < global_min)
       global_min = local_min(j);

CSCE 4610: Computer Architecture

More examples

```c
sum = 0.0;
for (i=0; i<n; i++)
    sum = sum +a[i];
```

Assume a separate thread computes each loop iteration.

Can we allow independent processors to modify sum directly?

```c
sum = 0.0;
for (i=0; i<n; i++)
{
    lock;
    sum = sum +a[i];
    unlock;
}
```
Barrier synchronization.

Most parallel algorithms look like the following:

- Fork threads
- Execute in parallel
- Join or Rendezvous at the barrier

None of the forked threads are allowed to proceed beyond the barrier. In some languages only the last thread to arrive at the barrier is allowed to proceed — all the other threads simply quit.

```c
{...
  Join();
  ...
  Exit;
}
```

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:

- Parallel Processing 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

\[ \text{loop level parallelism} = \text{data parallelism} \]

\[ \#pragma \text{omp parallel for} \]

\[ \text{for} \ (i=0; \ i<n; \ i++) \]

\{ loop body \}

We can execute each iteration in parallel (if there are no dependencies among iterations)

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, Rc, 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

Cray-1 (1976)
CSCE 4610: Computer Architecture

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

### 4 Instruction Buffers

### 64-bitx16 Vector Code Example

**# 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
```

### CSCE 4610: Computer Architecture

**Vector Code Example**

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

### Memory bank cycle 50 ns
### Processor cycle 12.5 ns (80MHz)
CSCE 4610: Computer Architecture

Text uses instructions similar to Cray but calls them Vector MIPS (VMIPS) -- see page 266

### Instruction Operators Function

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operators</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV_D</td>
<td>V1, V2, V3</td>
<td>Add elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>ADDS_D</td>
<td>V1, V2, F0</td>
<td>Add F0 to each element of V2, then put each result in V1.</td>
</tr>
<tr>
<td>SUBV_D</td>
<td>V1, V2, F0</td>
<td>Subtract elements of V2 from V1, then put each result in V1.</td>
</tr>
<tr>
<td>SUBS_D</td>
<td>V1, V2, F0</td>
<td>Subtract elements of V2 from F0, then put each result in V1.</td>
</tr>
<tr>
<td>MULV_D</td>
<td>V1, V2, V3</td>
<td>Multiply elements of V2 and V3, then put each result in V1.</td>
</tr>
<tr>
<td>MULS_D</td>
<td>V1, V2, F0</td>
<td>Multiply each element of V2 by F0, then put each result in V1.</td>
</tr>
<tr>
<td>DIVV_D</td>
<td>V1, V2, V3</td>
<td>Divide elements of V2 by V1, then put each result in V1.</td>
</tr>
<tr>
<td>DIVS_D</td>
<td>V1, V2, F0</td>
<td>Divide F0 by elements of V2, then put each result in V1.</td>
</tr>
<tr>
<td>LV</td>
<td>V1, R1</td>
<td>Load vector register V1 from memory starting at address R1.</td>
</tr>
<tr>
<td>SY</td>
<td>R1, V1</td>
<td>Store vector register V1 into memory starting at address R1.</td>
</tr>
<tr>
<td>UWE</td>
<td>V1, R1, R2</td>
<td>Load X1 from address at R1 with stride in R2 (i.e., R1 + i * R2).</td>
</tr>
<tr>
<td>SVC</td>
<td>(R1, R2), V1</td>
<td>Store V1 to address at R1 with stride in R2 (i.e., R1 + i * R2).</td>
</tr>
<tr>
<td>LV1</td>
<td>V1, (R3)+V2</td>
<td>Load V1 with vector whose elements are at R3 + V2 (i.e., R2 is an index).</td>
</tr>
<tr>
<td>SV1</td>
<td>(R3)+V2, V1</td>
<td>Store V1 to vector whose elements are at R3 + V2 (i.e., R2 is an index).</td>
</tr>
<tr>
<td>C1</td>
<td>V1, R1, R2</td>
<td>Create an index vector by storing the values 0, 1, R1 + 8, R1 + 16, ..., R1 + 8 * (R2 - 1) into V1.</td>
</tr>
<tr>
<td>S-&gt;V1, D</td>
<td>V1, R2</td>
<td>Compare the elements (V0, V6, V7, V8, V9, V10) 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-&gt;V1, D performs the same compare but using a scalar value as one operand.</td>
</tr>
<tr>
<td>POP</td>
<td>R1, VM</td>
<td>Count the 1s in vector-mask register VM and store count in R1.</td>
</tr>
<tr>
<td>CNW</td>
<td>Set the vector-mask register to all 1s.</td>
<td></td>
</tr>
<tr>
<td>MTC</td>
<td>V2, R1</td>
<td>Move contents of R1 to vector-length register V2.</td>
</tr>
<tr>
<td>MFC</td>
<td>R1, V1</td>
<td>Move the contents of vector-length register V1 to R1.</td>
</tr>
<tr>
<td>MFR</td>
<td>VM, F0</td>
<td>Move contents of F0 to vector-mask register VM.</td>
</tr>
<tr>
<td>MFR</td>
<td>F0, VM</td>
<td>Move contents of vector-mask register VM to F0.</td>
</tr>
</tbody>
</table>

In standard MIPS instructions we use a loop

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

CSCE 4610: Computer Architecture

Also let us look at how to use the vector instructions to compute the following vector operation

\[ Y = a \times X + Y \]

where X and Y are arrays (vectors)

In C

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

CSCE 4610 April 13, 2017
CSCE 4610: Computer Architecture

Note that all functional units are pipelined
   Vector-Vector units
   Vector-Scalar unit
   Load and Store Vector Units

In most vector processors, we can “forward” data from one vector operation to another.

For example we can forward the data from MULSV to ADDVV
Or, as we compute a*X[i] this can be forwarded so that we can do Y[i]+a*X[i]

Such forwarding is known as “Chaining” – we are connecting the output of one pipelined functional unit to another.

How do we estimate the performance of Vector processors?

Convoys and Chimes

CSCE 4610: Computer Architecture

Convey is a set of vector instructions that can be issued together
   A vector instruction cannot be issued if there is a structural hazard
   We will assume that data dependencies (RAW) can be handled with chaining
   And other data hazards with register renaming.

The time it takes to complete a convoy is known as a chime.

Consider the example we have seen already (and ignore the scalar load)

   LD     F0, a       : load scalar a into F0
   LV     V1, Rx     : load vector X into V1
   MULSV  V2, V1, F0 : compute a*X
   LV     V3, Ry     : load vector Y into V3
   ADDVV  V4, V2, V3 : compute a*X + Y
   SV     V4, Ry     : store Y

We have 3 convoys (ignoring scalar load)

   1.   LV V1, Rx     MULSV V2, V1, F0
   2.   LV V3, Ry     ADDVV V4, V2, V3
   3   SV

CSCE 4610 April 13, 2017
Since we have only one Load/Store vector unit, we have structural hazards.

The code takes 3 chimes.
How many operations did we complete (in total) = 5
How many floating point operations = 2

Or we achieved 1.5 chimes per FLOP.

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