CSCE 5610: Computer Architecture

Homework # 6 Using Simplescalar
Due November 8, 2018

Exam 2: Thursday Nov. 15
Help Session Tuesday 11/13 at 8am in F 280

Final Exam: Dec. 13, 10:30-12:30
Project Reports Due: Dec 6, 2018 (last day of classes)

Review
Dynamic scheduling of instructions: out of order execution
Scoreboard
Tomasulo’s technique (using reservation stations)
Speculative Execution
Re-Order Buffers
Multiple issue architectures

Speculative execution
We can issue and execute instructions speculatively based on branch prediction
But we must be able to “undo” execution of instructions
We need to buffer results of instructions and NOT MODIFY registers
Dependent registers can still use results from speculative instructions
Note the addition of “reorder buffers”
They store results of speculative instructions

From Text pages 208-217
Let us examine a slightly different example from Text on page 226.

Loop:

```
ld    x2, 0(x1)
addi  x2, x2, 1
sd    x2, 0(x1)
addi  x1, x1, 8
bne   x2, x3, Loop
```

Note: I have been using addi x1, x1, #8 to indicate immediate. Textbook (and RISC V) does not use # to indicate a constant.

First without speculation. Note that the ld following the bne cannot start execution earlier because it must wait until the branch outcome is determined.
With Speculation. Instructions beyond bne execute without waiting for branch decision.

Should you speculate across more than one branch – more complex hardware?

Speculate on both bz and bne branches

Note, now we are going beyond a basic block
CSCE 5610: Computer Architecture

Another type of multiple issue architecture is known as Very Large Instruction Word (VLIW) computer.

Basically, a single instruction word contains multiple instructions. There is some discussion in Chapter 3 pages 218-222.

When we fetch one such very long word instruction, we are actually fetching 5 instructions. Assuming that we have 5 different functional units (one Integer, 2 FP and two LD/ST), we can issue 5 instructions and then execute them.

There are several issues to consider:
- These 5 instructions must be independent
- Since we have different latencies, the next wide instruction must be properly scheduled

Consider an example loop:

```
fld f0, 0(x1)
fadd f4, f0, f2
fsd f4, 0(x1)
addi x1, x1, 8
blt x1, x2, loop
```

Unroll the loop 7 times

```
loop:  fld   f0, 0(x1)  loop:  fld   f0, 0(x1)
       fadd f4, f0, f2  fadd f10, 16(x1)
       fsd f4, 0(x1)  fsd f14, 24(x1)
       addi x1, x1, 8  addi f18, 32(x1)
       blt x1, x2, loop  blt f22, 40(x1)
```

```
addi x1, x1, 8  addi f26, 48(x1)
fsd f4, 0(x1)  fadd f12, f10, f2
fsd f14, 24(x1)  fadd f16, f14, f2
fsd f18, 32(x1)  fadd f20, f18, f2
fsd f22, 40(x1)  fadd f24, f22, f2
fsd f26, 48(x1)  fadd f28, f26, f2
addi x1, x1, 8
addi f28, 48(x1)
blt x1, x2, loop
```
CSCE 5610: Computer Architecture

<table>
<thead>
<tr>
<th>Integer</th>
<th>fp-1</th>
<th>fp-2</th>
<th>mem-1</th>
<th>mem-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fld f0, 0(x1)</td>
<td>fld f0, 8(x10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>fadd f4, f0, f2</td>
<td>fadd f8, f6, f2</td>
<td>fld f10, 16(x1)</td>
<td>fld f14, 24(x1)</td>
</tr>
<tr>
<td>3</td>
<td>fadd f12, f10, f2</td>
<td>fadd f16, f14, f2</td>
<td>fld f18, 32(x1)</td>
<td>fld f22, 40(x1)</td>
</tr>
<tr>
<td>4</td>
<td>fadd f20, f18, f2</td>
<td>fadd f24, f22, f2</td>
<td>fld f26, 48(x1)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>fadd f28, f26, f2</td>
<td></td>
<td>fsd f4, 0(x1)</td>
<td>fsd f8, 8(x1)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>fsd f12, 16(x1)</td>
<td>fsd f16, 24(x1)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>fsd f20, 32(x1)</td>
<td>fsd f24, 40(x1)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>addi x, x1, 56</td>
<td></td>
<td>fsd f28, -8(x1)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>blt x1, x2, loop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note we have 9 VLIW instructions. But we executed 23 operations so we have an average of 2.5 instructions per cycle

What is the difference between this and our out-of-order superscalar processor?

In VLIW, hardware does not change the order of instruction execution

Compiler needs to take care of unrolling and instruction scheduling

Less complex hardware

CSCE 5610  Nov. 06, 2018

How can compiler help in improving performance?

Unrolling loops. We have seen examples previously, but I want to introduce it again to talk about a new concept called register renaming

Consider the following example

LOOP:                      LOOP:
fld f2, 100 (x2)          fld f2, 100 (x2)
fadd f4, f2, f8           fadd f4, f2, f8
fmul f6, f4, f10          fmul f6, f4, f10
fsd f6, 100 (x2)          fsd f6, 100 (x2)
addi x2, x2, #8           addi x2, x2, #16
bne x2, x6, Loop          bne x2, x6, Loop

Unroll the loop twice

This causes some unnecessary WAR and WAW dependencies

We can eliminate them with “register renaming”

CSCE 5610  Nov. 06, 2018
By using new registers for the second iteration, we can eliminate WAR and WAW dependencies.

Let us see how many cycles are needed after reordering:

We have 26 cycles to complete 2 iterations = 13 cycles per iteration.

We can always eliminate WAR and WAW using "register renaming" if we have enough registers.

You CANNOT eliminate RAW "true dependency".

Can compiler eliminate these dependencies?

Compilers use different terms for RAW, WAR and WAW dependencies.

RAW: true dependency
WAR: Anti dependency
WAW: Output dependency

We can eliminate False dependencies by renaming --called Static Single Assignment.

In general this is more complicated when we have conditional statements.
Register renaming in hardware

In order to achieve out of order execution (and even speculation) we need to change registers

Name changing or register renaming can be done in software (compilers) or by hardware

Most processors have many more physical registers than logical registers
users can only use logical registers (32 or 64)

We can maintain state with each physical register (busy or free)
Note every time you change the register number, you need make changed to reflect RAW

Compiler Scheduling to aid ILP ---- Software Pipelining – particularly useful with VLIW

Consider the code

```
Loop:  fld  f0, 0(x1)
fadd f4, f0, f2
fsd  f4, 0(x1)
addi x1, x1, - 8
bnez x1, Loop
```

Consider the dependencies

\( \text{fsd needs the results form fadd and} \)
\( \text{fadd needs the value read by fld} \)

If we have say 2 cycle latency from fld to fadd, 3 from fadd to fsd how can we avoid these latencies?

We can unroll and schedule.
But unrolling cause code expansion larger instruction caches are needed. Need more registers
But consider a different way of thinking
CSCE 5610: Computer Architecture

We will take fld from i+2, fadd from i+1 and fsd from i-th iteration
And make these instructions (from different iterations) as a loop body

Init:  
  fld x6, 0(x1)  /* Preamble
  fld f0, -8(x1)
  fadd f4, f6, f2
  addi x1, x1, -16

Loop:  
  fsd f4, 16(x1)  /* fsd from i
  fadd f4, f0, f2  /* fadd from i+1
  fld f0, 0(x1)  /* fld from i+2
  addi x1, x1, -8
  bnez x1, Loop

Final:  
  fsd f4, 8(x1)  /* Post-Script
  fadd f4, f0, f2
  fsd f4, 0(x1)
CSCE 5610: Computer Architecture

What is the advantage? We can schedule the instructions better and not have stalls inside a loop. Let us only look at the instructions in the new loop body (ignore preamble and postscript)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Read Inputs</th>
<th>Execute</th>
<th>CDB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsd</td>
<td>1</td>
<td>2</td>
<td>3-4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
<td>2</td>
<td>3-6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>fld</td>
<td>1</td>
<td>3</td>
<td>4-6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>4</td>
<td>5-6</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>bnez</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>fsd</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>fadd</td>
<td>2</td>
<td>7</td>
<td>8-9</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>fld</td>
<td>2</td>
<td>8</td>
<td>9-10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>addi</td>
<td>2</td>
<td>9</td>
<td>10-11</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>bnez</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>fsd</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

fsd In iteration 2 reads data from fadd in iteration 1 (cycle 7)
fadd in iteration 2 reads data from fld in iteration 1 (cycle 8)
fld in iteration 2 reads data from addi in iteration 1 (cycle 9)

Another example of **Software Pipelining**

Consider the code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Read Inputs</th>
<th>Execute</th>
<th>CDB</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld</td>
<td>1</td>
<td>2</td>
<td>3-4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
<td>2</td>
<td>3-6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>fmul</td>
<td>1</td>
<td>3</td>
<td>4-6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>fsd</td>
<td>1</td>
<td>4</td>
<td>5-6</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>addi</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>fadd</td>
<td>2</td>
<td>7</td>
<td>8-9</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>fld</td>
<td>2</td>
<td>8</td>
<td>9-10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>addi</td>
<td>2</td>
<td>9</td>
<td>10-11</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>bnez</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>fsd</td>
<td>2</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

Consider the dependencies

fsd needs the results form fmul
fmul needs results from fadd
and 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 form fmul to fsd

how can we avoid these latencies
CSCE 5610: Computer Architecture

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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsd</td>
<td>f4, 24(x1)</td>
<td>fld uses data from fmul from previous iteration, there is 5 cycle delay</td>
</tr>
<tr>
<td>fmul</td>
<td>f6, f4, f4</td>
<td>Likewise fmul depends on fadd from previous iteration</td>
</tr>
<tr>
<td>fadd</td>
<td>f4, f0, f2</td>
<td></td>
</tr>
<tr>
<td>fld</td>
<td>f0, 0(x1)</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>x1, x1, -8</td>
<td>fld depends on fld from previous iteration</td>
</tr>
<tr>
<td>bne</td>
<td>x1, Loop</td>
<td></td>
</tr>
</tbody>
</table>

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

CSCE 5610: Computer Architecture

Instruction Level Parallelism is limited

Let us start with the maximum possible parallelism at instruction level (ILP)

Ideal Processor:
- Infinite fetch and decode bandwidth
- Instruction window refers to the set of instructions that can be examined for issue
- Infinite issue width
  - Infinite registers for renaming (or reorder buffers)
- Branch prediction is perfect
- Jump prediction is perfect (even dynamic branches are predicted)
- Memory addresses are known so that loads and stores can be reordered (unless they are accessing the same address)
Floating point benchmarks have higher ILP – Why?

More realistic situations

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
here the different data for each benchmarks shows how many instructions can you examine at a time (window) to determine on which instructions can be issued.
For some benchmarks the parallelism is very small (10 instructions).

Figure 3.2: The effect of window size shown by each application by plotting the number of instructions per clock cycle.

---

For most integer benchmarks, you get very small ILP (2-3) for floating point higher ILP is possible.
We can improve the ILP with loop level optimizations.
Another way to increase ILP is to execute multiple threads simultaneously.

What is difference between threads, tasks or processes?

Threads within a process (or task) share “resources”
- Share the memory
- Share signals
- Share files

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

---
CSCE 5610: Computer Architecture

Context switch can be very fast (2-4 cycles) since we need only to change the register context that is accessed by CPU.

Note, however, additional register sets add to hardware complexity. If designed as a single register file, need more read and write ports.

If designed as separate register files, “wire lengths” slows clock

When to switch between the threads?
- A thread is blocked due to a pending resource request
- A cache miss
- Or switch on every instruction.

SMT - Simultaneous multithreading fetches instructions from different threads and issues all the instructions from multiple threads together

Consider a simple pipelined processor with 5 stages (typical MIPS like)
And 5 threads labeled Th1 through Th5

<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>

Do we need branch prediction?
Thread Level Parallelism – computing performance

- Simple model
  - 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 \( U_i = \frac{R}{R+L} \)
CSCE 5610: Computer Architecture

- 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

- Maximum Utilization that is possible:
  - $U_{\text{max}} = \frac{R}{R+C}$
    - $C$ 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 $U_N = \frac{N \times R}{R+L}$