CSE 451: Operating Systems
Spring 2020

Module 0
(Instruction Level) Parallelism (cont.)
Today: Overview

• Control hazards
  • Speculation

• False dependences
  • Renaming

• Superscalars
  • More general parallelism
(5-stage) Pipelining: Data Hazard

lw x4, 8(x2)
add x3, x3, x4
sw x3, 0(x2)

Instruction Fetch

PC

+4

Read Registers

Arithmetic/Logic Unit

Memory Read/Write

Register Write Back

lw x3, 0(x2)
add x3, x3, x4
sw x3, 0(x2)
(5-stage) Pipelining: Data Hazard

\[
\begin{align*}
\text{lw} & \quad x4, 8(x2) \\
\text{add} & \quad x3, x3, x4 \\
\text{sw} & \quad x3, 0(x2)
\end{align*}
\]
(5-stage) Pipelining: Control Hazards

beq \( x_3, x_4, \text{skip} \)
add \( x_3, x_4, x_5 \)

\text{skip:}
add \( x_7, x_3, x_6 \)

beq \( x_3, x_4, \text{skip} \)
(5-stage) Pipelining: Control Hazards

```
beq x3, x4, skip
add x3, x4, x5
skip:
    add x7, x3, x6
```

Is that the right next instruction?
Dealing with Control Hazards

• We can recognize that we have a branch in the cycle during which it is fetched, but...
• We won’t know whether it’s taken or not until the third cycle of its execution, and...
• We won’t have computed the target address until at least the second cycle of its execution
• What instruction should be fetched immediate after we have fetched the branch?
  • We don’t know!
Option 0: Pessimistic / Bubbles

```
beq x3, x4, skip
add x3, x4, x5
skip:
add x7, x3, x6
```

Con: Executing a branch always costs 3 instruction issue slots
Option 0+: Expose “branch delay slots” in ISA

• This is a variant of the general scheme “make it the next layer up’s problem” (also used in operating systems)

• Compiler understands that the next two instructions after a branch are always executed
  • tries to find instructions that “naturally” would come before branch and move the move them to after branch
  •  \(<\text{some instruction}>\)
     \begin{Verbatim}
     \text{beq} \ldots
     \end{Verbatim}
    becomes
    \begin{Verbatim}
     \text{beq} \ldots
    \end{Verbatim}
    \(<\text{some instruction}>\)
  • When is it legal for the compiler to do that?
  • If the compiler can’t find any instructions to fill the slots, it puts NOPs there
Option 1: Speculate #1: Assume Not Taken

Instruction Fetch

beq  x3, x4, skip
add  x3, x4, x5

skip:
add  x7, x3, x6

Read Registers

Arithmetic/Logic Unit

Memory Read/Write

Register Write Back

If we find out the branch is taken, we have to purge the two mis-fetched instructions
Speculate #2: Assume Taken

```
beq x3, x4, skip
add x3, x4, x5
skip:
    add x7, x3, x6
```

Don’t know the target address soon enough. This is bad...
Speculate #3: Branch Table Prediction

beq x3, x4, skip
add x3, x4, x5
skip:
  add x7, x3, x6

beq x3, x4, skip
add x3, x4, x5

Branch Table Maintenance

<table>
<thead>
<tr>
<th>PC</th>
<th>Next Inst Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3EF0804</td>
<td>0x3EF0808</td>
</tr>
<tr>
<td>0x28808</td>
<td>0x421FC0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Table is indexed by value of the PC
- When you fetch a branch, find a row for that branch
  - Enter its PC as the tag, if row doesn’t currently exist
  - When you figure out what the actual next instruction is, fill in the next address column of that row
- As you fetch instructions, use the PC’s value to look for a matching row in the table
  - If you find one, set the PC to the next inst address field
- That is, predict that the next time the branch is executed it will do the same thing it did the last time
  - “The future looks like the past”
Branches Due to Loops

• for (i=0; i<N; i++) { ...}  
• There’s a conditional branch at the bottom of the loop testing if i<N  
• The loop is taken each time it’s reached, except when i == N  
• That last iteration causes the branch table to predict not taken on the first iteration the next time the loop is reached  
• So, you get two mis-predictions each time the loop is executed  
• Can you do better?  
  • Sure. When you figure out the target address for a branch, predict taken if the branch is to lower memory addresses and not-taken if it’s to higher addresses  
    • The loop branch will always predict taken, which means there’s only one mis-prediction per loop execution
Can We Do Even Better?

• Maybe

• Branches show up for conditionals as well as loops
  • If...then...else, for instance

• Use a scheme that remembers the past, but is willing to change its mind
  • Add two bits per prediction table row to keep track of state
Control Hazards Summary

• When you’re trying to go fast, you can’t afford to wait
  • Long latency operations:
    • Decide whether or not a branch will be taken
    • Retrieve the contents of a web page
Control Hazards Summary
Moving Beyond Pipelines: Superscalars

• Pipelines have some important limitations
  • Only one instruction can be issued per cycle
    • Because of hazards, number of instructions completed per cycle will be less than one
  • A dependence that stalls one instruction necessarily stalls all instructions after it, even if they have no dependences themselves
    • This could be addressed to some extent by the compiler, assuming it understood the implementation of the datapath
  • Even if I add more hardware (e.g., more ALUs), I can’t use them
Superscalars

• Some of the following slides are from https://ece752.ece.wisc.edu/lect05-superscalar-org.pdf
What Does a High-IPC CPU Do?

1. Fetch and decode
2. Construct data dependence graph (DDG)
3. Evaluate DDG
4. Commit changes to program state

Source: [Palacharla, Jouppi, Smith, 1996]

Mikko Lipasti-University of Wisconsin
A Typical High-IPC Processor

1. Fetch and Decode

2. Construct DDG

3. Evaluate DDG

4. Commit results
Constructing the Dependence Graph

- The dependences are RAW, WAR, and WAW
- RAW is a “true dependence”
- WAR and WAW are “false dependences”
  - They’re false because they’re conflicts on names, not on values
- WAW:
  - \(\text{add } x_3, x_2, x_1\) \(\text{add } x_3, x_2, x_1\)
  - \(\text{add } x_3, x_4, x_5\) \(\text{add } x_50, x_4, x_5\)
- The compiler produces conflicts on names because the ISA has only so many registers (e.g., 16 or 32)
- The hardware implementation can have many more registers
  - The hardware does “register renaming” as it fetches instructions to break false dependences
Register Renaming

• Instead of thinking of this code as naming registers, think of it as naming values

  add x3, x2, x1       add <value3>, x2, x1
  ...

  vs.

  add x3, x4, x5       add <value6>, x2, x1

• These “value names” identify the true dependences (RAW)
• The hardware is free to map the value names to whatever physical registers it wants
Register Renaming

• When you assign a name (one of the hardware registers) to a value, you have to propagate it to future instructions

• Hardware keeps a table telling it which value name (hardware register) currently represents each architectural register name

• add x3, x2, x1  
  ... becomes  
  add 51, <x2>, <x1>

  ...  
  add x2, x3, x4  
  add 43, 51, <x4>  
  add x3, x7, x2  
  add 48, <x7>, 43

Hardware maintains a table with a row that says “x3 is 51” from the time it sees the first instruction until the time it sees the name “x3” refer to a different value
Register Renaming Summary

• WAW and WAR dependences can be eliminated by renaming
• That’s true for the CPU executing instructions that modify registers
• That’s true for software (including the OS) that updates variables/data structures
  • If multiple threads share a data structure, the code likely needs to explicitly synchronize their execution
    • Synchronization is an overhead, analogous to bubbles in the pipeline
• We “rename” by taking a centralized shared data structure and replacing it with many more similar data structures
  • Often, a “private” data structure per thread and synchronization code that manages the private instances and makes them act as a single instance