Digital Logic Design

234262

Alex Bronstein

Lecture 11: Instruction-level parallelism
Abstraction Hierarchy

- Application Software
  - Operating System *(threads, files, exceptions)*
  - Computer Architecture *(instruction set)*
  - Micro-Architecture *(execution pipeline)*
    - Logic *(adders, multipliers, FSMs)*
    - Digital Circuits *(gates)*
    - Analog Circuits *(amplifiers)*
    - Process and Devices *(MOS FET transistors)*
    - Physics *(electrons, holes, tunneling, band gap)*
Instruction-level parallelism

- Exploit parallelism among instructions!

- Deeper pipeline to allow more instructions to overlap
  - Break instruction to more stages
  - Run at a faster clock

- Multiple issue: launch multiple instructions at the same clock cycle
  - Static: scheduling done by the compiler
  - Dynamic: scheduling done at runtime by the processor

- Single instruction multiple data (SIMD): vector operations
  - Particularly useful when applying the same program to a stream of data such as in multimedia application
## Deeper pipelines

- **Deeper pipelines** and faster clocks

- Example of Intel processors:

<table>
<thead>
<tr>
<th></th>
<th>486</th>
<th>Pentium</th>
<th>Pentium Pro</th>
<th>Pentium IV</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>’89</td>
<td>’93</td>
<td>’97</td>
<td>‘01–’04</td>
<td>’06</td>
</tr>
<tr>
<td><strong>Stages</strong></td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>22–31</td>
<td>14</td>
</tr>
<tr>
<td><strong>Clock [Hz]</strong></td>
<td>25M</td>
<td>60M</td>
<td>200M</td>
<td>2–3.6G</td>
<td>2.9G</td>
</tr>
<tr>
<td><strong>Power [W]</strong></td>
<td>5</td>
<td>10</td>
<td>30</td>
<td>75–100</td>
<td>75</td>
</tr>
</tbody>
</table>

- Large branch penalties

- Fast clocks are power inefficient

- Modern architectures favor multiple lower-frequency cores
Static multiple issue

- Compiler packages instructions into **issue slots**
- Several instructions are launched in a single clock cycle
- More hardware is typically required inside the CPU core
- **Very long instruction word** (VLIW) architectures
- Common in DSP and embedded processors
Very long instruction word (VLIW)

- Example: static two-issue MIPS (as in some embedded MIPS variants)
- One 64-bit instruction packet with two issue slots:

```
| 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
```

| Load/Store instruction | ALU/Branch instruction |

- IMEM is 64-bit aligned
- Unused slots must be filled with nops
# Static two-issue pipeline

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>N+4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>N+8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>N+12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>N+16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>N+20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Regular one-issue pipeline (simplified)
Static two-issue pipeline
Static multiple issue

- IMEM has 64-bit data bus to read 2 32-bit instructions from a packet in a single cycle
- Register file has 2 additional read ports (4 total) and one additional write port (2 total)
- Separate adder needed to compute data memory address
- Twice more instructions execute in parallel (ideally)
Static multiple issue

- EX data hazard
  - add $t0, $s0, $s1
  - lw $s2, 0($t0)

- Forwarding avoided stalls in single-issue pipeline

- Cannot forward ALU result to Load/Store in the same packet

- Still one cycle load-use latency, but now next two instructions cannot use load result without stalling

- More sophisticated scheduling is needed!
for (int* p = A; p>0; p--) {
    *p += offset;
}

# $s1 = A
# $s2 = offset

loop:  lw  $t0, 0($s1)  # t0 = array element
        addu $t0, $t0, $s2  # add scalar in $s2
        sw  $t0, 0($s1)    # store result
        addi $s1, $s1, -4  # decrement pointer
        bne  $s1, $zero, loop  # repeat while $s1!=0
Data dependencies

# One-issue - 5 cycles/iteration

loop:  lw  $t0, 0($s1)
        addu  $t0, $t0, $s2
        sw  $t0, 0($s1)
        addi  $s1, $s1, -4
        bne  $s1, $zero, loop

# Two-issue - 4 cycles/iteration

loop:  nop  |  lw  $t0, 0($s1)
        addi  $s1, $s1, -4  |  nop
        addu  $t0, $t0, $s2  |  nop
        bne  $s1, $zero, loop  |  sw  $t0, 4($s1)
// Original loop
for (int* p = A; p>0; p--) {
    *p += offset;
}

// Unrolled in ‘bunches’ of 4 - less branch overhead
for (int* p = A; p>0; p-=4) {
    *p    += offset;
    *(p+1) += offset;
    *(p+2) += offset;
    *(p+3) += offset;
}
Loop unrolling

# Unrolled with name dependences - 13 cycles/4 iters
loop:
  nop
  lw $t0, 0($s1)
  addu $t0, $t0, $s2
  nop
  sw $t0, 0($s1)
  lw $t0, -4($s1)
  addu $t0, $t0, $s2
  nop
  sw $t0, -4($s1)
  lw $t0, -8($s1)
  addu $t0, $t0, $s2
  nop
  sw $t0, -8($s1)
  lw $t0, -12($s1)
  addu $t0, $t0, $s2
  nop
  addi $s1, $s1, -16
  sw $t0, -12($s1)
  bne $s1, $zero, loop
  nop
Name dependences

- **Name dependence** (aka **antidependence**): an ordering forced by the reuse of a name (typically of a register) rather than by true dependence.

- **Register renaming**: the renaming of the registers (by compiler or hardware) to remove name dependences.
Unrolled with renaming - 8 cycles/4 iterations

```
loop:  addi   $s1, $s1, -16  | lw   $t0, 0($s1)
      nop                  | lw   $t1, 12($s1)
      addu  $t0, $t0, $s2  | lw   $t2, 8($s1)
      addu  $t1, $t1, $s2  | lw   $t3, 4($s1)
      addu  $t2, $t2, $s2  | sw   $t0, 16($s1)
      addu  $t3, $t2, $s2  | sw   $t1, 12($s1)
      nop                  | sw   $t2, 8($s1)
      bne  $s1, $zero, loop | sw   $t3, 4($s1)
```
Dynamic multiple issue

- Instruction scheduled into issue slots **dynamically at runtime**
- Compiler tries to place instructions a priori into a beneficial order
- **Superscalar** architectures dynamically select instructions to issue based on dependencies or speculations
- **Out-of-order** architectures further allow a blocked instruction not to cause the following instructions to wait
- Common in many modern high-performance CPUs
Single instruction multiple data (SIMD)

- Example: MIPS SIMD architecture
- 32 128-bit vector registers addressed by special instructions
- Each 128-bit register can be interpreted as
  - 4-vector of 32-bit words
    
    | W3 | W2 | W1 | W0 |
    |----|----|----|----|
  - 8-vector of 16-bit half-words
    
    | H7 | H6 | H5 | H4 | H3 | H2 | H1 | H0 |
    |----|----|----|----|----|----|----|----|
  - 16-vector of bytes
    
    | BF | BE | BD | BC | BB | BA | B9 | B8 | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
    |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
- Single instruction operates on the entire vector
Single instruction multiple data (SIMD)

Syntax: \texttt{addv.w \, wd, \, ws, \, wt}
Semantics: element-wise word vector addition
Single instruction multiple data (SIMD)

**Syntax:**  
`fill.w wt, rs`

**Semantics:** fills all vector elements with the content of a GPR
Single instruction multiple data (SIMD)

Syntax: \( \text{ld.w} \ \text{wd, imm}(\text{rs}) \)

Semantics: reads vector contents from memory
Single instruction multiple data (SIMD)

Syntax: \texttt{st.w wd, imm(rs)}
Semantics: stores vector contents to memory

\begin{itemize}
  \item Syntax: \texttt{st.w wd, imm(rs)}
  \item Semantics: stores vector contents to memory
\end{itemize}