Chapter 4

The Processor: C
Multiple Issue
Based on P&H
Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel

- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice
Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime
Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing

- Common to static and dynamic multiple issue

- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation
Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check

- Static speculation
  - Can add ISA support for deferring exceptions

- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - \( \Rightarrow \) Very Long Instruction Word (VLIW)
Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</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>
MIPS with Static Dual Issue

Chapter 4 — The Processor — 10
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - $t0, $s0, $s1
    - $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
Scheduling Example

- Schedule this for dual-issue MIPS

```
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 # branch $s1!=0
```

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name
**Loop Unrolling Example**

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,-16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
- Closer to 2, but at cost of registers and code size
Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order

Example

```
lw   $t0, 20($s2)  
addu $t1, $t0, $t2  
sub  $s4, $s4, $t3  
slti $t5, $s4, 20  
```

- Can start sub while addu is waiting for lw
Dynamically Scheduled CPU

- Instruction fetch and decode unit
- Reservation station
- Reservation station
- Reservation station
- Reservation station
- Functional units
- Integer
- Integer
- Floating point
- Load-store

- In-order issue
- Preserves dependencies
- Hold pending operands
- Out-of-order execute
- Results also sent to any waiting reservation stations
- In-order commit
- Can supply operands for issued instructions
- Commit unit
- Reorders buffer for register writes

Chapter 4 — The Processor — 17
Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required
Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined

- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
Does Multiple Issue Work?

**The BIG Picture**

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>
## Cortex A8 and Intel i7

<table>
<thead>
<tr>
<th>Processor</th>
<th>ARM A8</th>
<th>Intel Core i7 920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Market</td>
<td>Personal Mobile Device</td>
<td>Server, cloud</td>
</tr>
<tr>
<td>Thermal design power</td>
<td>2 Watts</td>
<td>130 Watts</td>
</tr>
<tr>
<td>Clock rate</td>
<td>1 GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>Cores/Chip</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Floating point?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple issue?</td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Peak instructions/clock cycle</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Pipeline schedule</td>
<td>Static in-order</td>
<td>Dynamic out-of-order with speculation</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>2-level</td>
<td>2-level</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; level caches/core</td>
<td>32 KiB I, 32 KiB D</td>
<td>32 KiB I, 32 KiB D</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt; level caches/core</td>
<td>128-1024 KiB</td>
<td>256 KiB</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt; level caches (shared)</td>
<td>-</td>
<td>2- 8 MB</td>
</tr>
</tbody>
</table>
ARM Cortex-A8 Performance

Chapter 4 — The Processor — 25
Core i7 Pipeline
Core i7 Performance

![Graph showing CPI and misprediction percentages for various benchmarks.](image)

Chapter 4 — The Processor — 27
Matrix Multiply

Unrolled C code

```c
#include <x86intrin.h>
#define UNROLL (4)

void dgemm (int n, double* A, double* B, double* C)
{
    for ( int i = 0; i < n; i+=UNROLL*4 )
    for ( int j = 0; j < n; j++ ) {
        __m256d c[4];
        for ( int x = 0; x < UNROLL; x++ )
            c[x] = _mm256_load_pd(C+i+x*4+j*n);
        for( int k = 0; k < n; k++ )
            __m256d b = _mm256_broadcast_sd(B+k+j*n);
            for (int x = 0; x < UNROLL; x++)
                c[x] = _mm256_add_pd(c[x],
                _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
        }
    for ( int x = 0; x < UNROLL; x++ )
        _mm256_store_pd(C+i+x*4+j*n, c[x]);
}
```
Matrix Multiply

Assembly code:

1. `vmovapd (%r11),%ymm4` # Load 4 elements of C into %ymm4
2. `mov %rbx,%rax` # register %rax = %rbx
3. `xor %ecx,%ecx` # register %ecx = 0
4. `vmovapd 0x20(%r11),%ymm3` # Load 4 elements of C into %ymm3
5. `vmovapd 0x40(%r11),%ymm2` # Load 4 elements of C into %ymm2
6. `vmovapd 0x60(%r11),%ymm1` # Load 4 elements of C into %ymm1
7. `vbroadcastsd (%rcx,%r9,1),%ymm0` # Make 4 copies of B element
8. `add $0x8,%rcx` # register %rcx = %rcx + 8
9. `vmulpd (%rax),%ymm0,%ymm5` # Parallel mul %ymm1,4 A elements
10. `vaddpd %ymm5,%ymm4,%ymm4` # Parallel add %ymm5, %ymm4
11. `vmulpd 0x20(%rax),%ymm0,%ymm5` # Parallel mul %ymm1,4 A elements
12. `vaddpd %ymm5,%ymm3,%ymm3` # Parallel add %ymm5, %ymm3
13. `vmulpd 0x40(%rax),%ymm0,%ymm5` # Parallel mul %ymm1,4 A elements
14. `vmulpd 0x60(%rax),%ymm0,%ymm0` # Parallel mul %ymm1,4 A elements
15. `add %r8,%rax` # register %rax = %rax + %r8
16. `cmp %r10,%rcx` # compare %r8 to %rax
17. `jne 68 <dgemm+0x68>` # jump if not %r8 != %rax
18. `add $0x1,%esi` # register % esi = % esi + 1
19. `vmovapd %ymm4,(%r11)` # Store %ymm4 into 4 C elements
20. `vmovapd %ymm3,0x20(%r11)` # Store %ymm3 into 4 C elements
21. `vmovapd %ymm2,0x40(%r11)` # Store %ymm2 into 4 C elements
22. `vmovapd %ymm1,0x60(%r11)` # Store %ymm1 into 4 C elements

Chapter 4 — The Processor — 29
Performance Impact

![Graph showing performance impact with unoptimized, AVX, and AVX+unroll categories. The graph indicates a significant improvement with AVX+unroll compared to the unoptimized and AVX cases.]
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions
Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots
Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall