Unit 9: Superscalar Pipelines

Slides developed by M. Martin, A. Roth & C.J. Taylor at the University of Pennsylvania
with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.
A Key Theme: Parallelism

- Previously: pipeline-level parallelism
  - Work on execute of one instruction in parallel with decode of next

- Next: instruction-level parallelism (ILP)
  - Execute multiple independent instructions fully in parallel

- Then:
  - Static & dynamic scheduling
    - Extract much more ILP
  - Data-level parallelism (DLP)
    - Single-instruction, multiple data (one insn., four 64-bit adds)
  - Thread-level parallelism (TLP)
    - Multiple software threads running on multiple cores
This Unit: (In-Order) Superscalar Pipelines

- Idea of instruction-level parallelism
- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch
- “Superscalar” vs VLIW/EPIC
So far we have looked at **scalar pipelines**
- One instruction per stage
  - With control speculation, bypassing, etc.
    - Performance limit (aka “Flynn Bottleneck”) is CPI = IPC = 1
    - Limit is never even achieved (hazards)
    - Diminishing returns from “super-pipelining” (hazards + overhead)
An Opportunity...

• But consider:
  
  ADD r1, r2 -> r3
  ADD r4, r5 -> r6
  
  • Why not execute them *at the same time?* (We can!)

• What about:
  
  ADD r1, r2 -> r3
  ADD r4, r3 -> r6
  
  • In this case, *dependencies* prevent parallel execution

• What about three instructions at a time?
  • Or four instructions at a time?
What Checking Is Required?

- For two instructions: 2 checks
  
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks})
  \]

- For three instructions: 6 checks
  
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_3 \quad (4 \text{ checks})
  \]

- For four instructions: 12 checks
  
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_3 \quad (4 \text{ checks}) \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_4 \quad (6 \text{ checks})
  \]

- Plus checking for load-to-use stalls from prior \( n \) loads
What Checking Is Required?

- For two instructions: 2 checks
  
  \[
  \text{ADD src1}_1, \text{src2}_1 \rightarrow \text{dest}_1 \\
  \text{ADD src1}_2, \text{src2}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks})
  \]

- For three instructions: 6 checks
  
  \[
  \text{ADD src1}_1, \text{src2}_1 \rightarrow \text{dest}_1 \\
  \text{ADD src1}_2, \text{src2}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD src1}_3, \text{src2}_3 \rightarrow \text{dest}_3 \quad (4 \text{ checks})
  \]

- For four instructions: 12 checks
  
  \[
  \text{ADD src1}_1, \text{src2}_1 \rightarrow \text{dest}_1 \\
  \text{ADD src1}_2, \text{src2}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD src1}_3, \text{src2}_3 \rightarrow \text{dest}_3 \quad (4 \text{ checks}) \\
  \text{ADD src1}_4, \text{src2}_4 \rightarrow \text{dest}_4 \quad (6 \text{ checks})
  \]

- Plus checking for load-to-use stalls from prior $n$ loads
How do we build such “superscalar” hardware?
Multiple-Issue or “Superscalar” Pipeline

• Overcome this limit using **multiple issue**
  • Also called **superscalar**
  • Two instructions per stage at once, or three, or four, or eight...
  • “**Instruction-Level Parallelism (ILP)**” [Fisher, IEEE TC’81]

• Today, typically “4-wide” (Intel Core i7, AMD Opteron)
  • Some more (Power5 is 5-issue; Itanium is 6-issue)
  • Some less (dual-issue is common for simple cores)
A Typical Dual-Issue Pipeline (1 of 2)

- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte average instruction length)
  - Predict a single branch per cycle

- Parallel decode
  - Need to check for conflicting instructions
    - Is output register of $I_1$ is an input register to $I_2$?
  - Other stalls, too (for example, load-use delay)
A Typical Dual-Issue Pipeline (2 of 2)

- Multi-ported register file
  - Larger area, latency, power, cost, complexity
- Multiple execution units
  - Simple adders are easy, but bypass paths are expensive
- Memory unit
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a read port to data cache
    - Larger area, latency, power, cost, complexity
How Much ILP is There?

- The compiler tries to "schedule" code to avoid stalls
  - Even for scalar machines (to fill load-use delay slot)
  - Even harder to schedule multiple-issue (superscalar)

- How much ILP is common?
  - Greatly depends on the application
    - Consider memory copy
    - Unroll loop, lots of independent operations
  - Other programs, less so

- Even given unbounded ILP, superscalar has implementation limits
  - IPC (or CPI) vs clock frequency trade-off
  - Given these challenges, what is reasonable today?
    - ~4 instruction per cycle maximum
## Superscalar Pipeline Diagrams - Ideal

### Scalar

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw 0(r1) → r2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 4(r1) → r3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 8(r1) → r4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r14, r15 → r6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r12, r13 → r7</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r17, r16 → r8</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r18) → r9</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2-way superscalar

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw 0(r1) → r2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 4(r1) → r3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 8(r1) → r4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r14, r15 → r6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r12, r13 → r7</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r17, r16 → r8</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw 0(r18) → r9</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Superscalar Pipeline Diagrams - Realistic

**scalar**

1. `lw 0(r1) → r2`
2. `lw 4(r1) → r3`
3. `lw 8(r1) → r4`
4. `add r4, r5 → r6`
5. `add r2, r3 → r7`
6. `add r7, r6 → r8`
7. `lw 4(r8) → r9`

```
1  2  3  4  5  6  7  8  9 10 11 12
F D X M W
F D X M W
F D X M W
F D d* X M W
F D X M W
```

**2-way superscalar**

1. `lw 0(r1) → r2`
2. `lw 4(r1) → r3`
3. `lw 8(r1) → r4`
4. `add r4, r5 → r6`
5. `add r2, r3 → r7`
6. `add r7, r6 → r8`
7. `lw 4(r8) → r9`

```
1  2  3  4  5  6  7  8  9 10 11 12
F D X M W
F D X M W
F D X M W
F D d* d* X M W
F D d* X M W
F D X M W
F d* D X M W
```
Superscalar Implementation Challenges
Superscalar Challenges - Front End

- **Superscalar instruction fetch**
  - Modest: fetch multiple instructions per cycle
  - Aggressive: buffer instructions and/or predict multiple branches

- **Superscalar instruction decode**
  - Replicate decoders

- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - More complex stall logic - order $N^2$ for $N$-wide machine
  - Not all combinations of types of instructions possible

- **Superscalar register read**
  - Port for each register read (4-wide superscalar $\Rightarrow$ 8 read “ports”)
  - Each port needs its own set of address and data wires
    - Latency & area $\propto \#ports^2$
Superscalar Challenges - Back End

- **Superscalar instruction execution**
  - Replicate arithmetic units (but not all, for example, integer divider)
  - Perhaps multiple cache ports (slower access, higher energy)
    - Only for 4-wide or larger (why? only ~35% are load/store insn)

- **Superscalar bypass paths**
  - More possible sources for data values
  - Order \((N^2 \times P)\) for \(N\)-wide machine with execute pipeline depth \(P\)

- **Superscalar instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar \(\rightarrow\) 4 write ports

- **Fundamental challenge:**
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism
Superscalar Bypass

- **$N^2$ bypass network**
  - $N+1$ input muxes at each ALU input
  - $N^2$ point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load

- And this is just one bypass stage (MX)!
  - There is also WX bypassing
  - Even more for deeper pipelines

- One of the big problems of superscalar
  - Why? On the critical path of single-cycle “bypass & execute” loop
Not All $N^2$ Created Equal

- $N^2$ bypass vs. $N^2$ stall logic & dependence cross-check
  - Which is the bigger problem?

- $N^2$ bypass ... by far
  - 64-bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)

- Dependence cross-check not even 2nd biggest $N^2$ problem
  - Regfile is also an $N^2$ problem (think latency where $N$ is #ports)
  - And also more serious than cross-check
Mitigating $N^2$ Bypass & Register File

- **Clustering**: mitigates $N^2$ bypass
  - Group ALUs into $K$ clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - **With 1 or 2 cycle delay**
      - Can hurt IPC, but faster clock
  - $(N/K) + 1$ inputs at each mux
  - $(N/K)^2$ bypass paths in each cluster

- **Steering**: key to performance
  - Steer dependent insns to same cluster

- **Cluster register file**, too
  - Replicate a register file per cluster
  - All register writes update all replicas
  - Fewer read ports; only for cluster
Mitigating $N^2$ RegFile: Clustering++

- **Clustering**: split $N$-wide execution pipeline into $K$ clusters
  - With centralized register file, 2N read ports and N write ports

- **Clustered register file**: extend clustering to register file
  - Replicate the register file (one replica per cluster)
  - Register file supplies register operands to just its cluster
  - All register writes go to all register files (keep them in sync)
  - Advantage: fewer read ports per register!
    - $K$ register files, each with $2N/K$ read ports and N write ports
Another Challenge: Superscalar Fetch

- What is involved in fetching multiple instructions per cycle?
- In same cache block? → no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- What about taken branches?
  - How many instructions can be fetched on average?
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4, which is bad)
Increasing Superscalar Fetch Rate

• Option #1: over-fetch and buffer
  • Add a queue between fetch and decode (18 entries in Intel Core2)
  • Compensates for cycles that fetch less than maximum instructions
  • “decouples” the “front end” (fetch) from the “back end” (execute)

• Option #2: “loop stream detector” (Core 2, Core i7)
  • Put entire loop body into a small cache
    • Core2: 18 macro-ops, up to four taken branches
    • Core i7: 28 micro-ops (avoids re-decoding macro-ops!)
  • Any branch mis-prediction requires normal re-fetch

• Other options: next-next-block prediction, “trace cache”
Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - **What we’ve talked about thus far**
    + Executes unmodified sequential programs
    - Hardware must figure out what can be done in parallel
  - **E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)**

- **Very Long Instruction Word (VLIW)**
  - **Compiler identifies independent instructions**, new ISA
    + Hardware can be simple and perhaps lower power
  - **E.g., TransMeta Crusoe (4-wide)**
  - **Variant: Explicitly Parallel Instruction Computing (EPIC)**
    - A bit more flexible encoding & some hardware to help compiler
    - **E.g., Intel Itanium (6-wide)**

- **Dynamically-scheduled superscalar (next topic)**
  - **Hardware extracts more ILP by on-the-fly reordering**
  - **Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)**
## Trends in Single-Processor Multiple Issue

<table>
<thead>
<tr>
<th>Year</th>
<th>486</th>
<th>Pentium</th>
<th>PentiumII</th>
<th>Pentium4</th>
<th>Itanium</th>
<th>ItaniumII</th>
<th>Core2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - Not enough ILP to justify going to wider issue
  - Hardware or compiler *scheduling* needed to exploit 4-6 effectively
    - More on this in the next unit

- For high-performance *per watt* cores (say, smart phones)
  - Typically 2-wide superscalar (but increasing each generation)
Multiple Issue Redux

- Multiple issue
  - Exploits insn level parallelism (ILP) beyond pipelining
  - Improves IPC, but perhaps at some clock & energy penalty
  - 4-6 way issue is about the peak issue width currently justifiable
    - Low-power implementations today typically 2-wide superscalar

- Problem spots
  - $N^2$ bypass & register file $\rightarrow$ clustering
  - Fetch + branch prediction $\rightarrow$ buffering, loop streaming, trace cache
  - $N^2$ dependency check $\rightarrow$ VLIW/EPIC (but unclear how key this is)

- Implementations
  - Superscalar vs. VLIW/EPIC
This Unit: (In-Order) Superscalar Pipelines

- Idea of instruction-level parallelism
- Superscalar hardware issues
  - Bypassing and register file
  - Stall logic
  - Fetch
- “Superscalar” vs VLIW/EPIC
Trends in Single-Processor Multiple Issue

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - No justification for going wider
  - Hardware or compiler “scheduling” needed to exploit 4-6 effectively

- For high-performance *per watt* cores, issue width is ~2
  - Advanced scheduling techniques not needed
  - Multi-threading (a little later) helps cope with cache misses
Impact of Branch Prediction

- Base CPI for scalar pipeline is 1
- **Base CPI for N-way superscalar pipeline is 1/N**
  - Amplifies stall penalties
  - Assumes no data stalls (an overly optimistic assumption)

- Example: Branch penalty calculation
  - 20% branches, 75% taken, 2 cycle penalty, no branch prediction
  - Scalar pipeline
    - $1 + 0.2 \times 0.75 \times 2 = 1.3 \rightarrow 1.3/1 = 1.3 \rightarrow 30\%$ slowdown
  - 2-way superscalar pipeline
    - $0.5 + 0.2 \times 0.75 \times 2 = 0.8 \rightarrow 0.8/0.5 = 1.6 \rightarrow 60\%$ slowdown
  - 4-way superscalar
    - $0.25 + 0.2 \times 0.75 \times 2 = 0.55 \rightarrow 0.55/0.25 = 2.2 \rightarrow 120\%$ slowdown
Predication

- Branch mis-predictions hurt more on superscalar
  - Replace difficult branches with something else...
  - Convert control flow into data flow (& dependencies)
  - Helps hard-to-predict branches (but can hurt predictable branches)

- **Predication**
  - Conditionally executed insns unconditionally fetched
  - **Full predication** (ARM, Intel Itanium)
    - Can tag every insn with predicate, but extra bits in instruction
  - **Conditional moves** (Alpha, x86)
    - Construct appearance of full predication from one primitive
      
```cmoveq r1,r2,r3  // if (r1==0) r3=r2;
```
    - May require some code duplication to achieve desired effect
    - Doesn’t handle conditional memory operations
  + Only good way of adding predication to an existing ISA

- **If-conversion**: replacing control with predication
Predication If-Conversion Example

**Source code**

A = Y[i];
if (A == 0)
    A = W[i];
else
    Y[i] = 0;
Z[i] = A*X[i];

**Machine code**

0: ldf Y(r1),f2
1: fbne f2,4
2: ldf W(r1),f2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)

**Using Predication**

0: ldf Y(r1),f2
1: fspne f2,p1
2: ldf.p p1,W(r1),f2
3: jump 5
4: stf.np p1,f0,Y(r1)
5: ldf X(r1),f4
6: mulf f4,f2,f6
7: stf f6,Z(r1)
ISA Support for Predication

• Itanium: change branch 1 to **set-predicate insn fspne**
• Change insns 2 and 4 to **predicated insns**
  • *ldf.p* performs *ldf* if predicate *p1* is true
  • *stf.np* performs *stf* if predicate *p1* is false
CMOV Prediction Example

x86 only has a “CMOV” instruction
  • Note: in x86’s CMOV, any “load” part is non-conditional
  • Small change in the code helps the compiler optimize
Another CMOV Example (Part I)

- gcc -Os -fno-if-conversion

```c
#include <stdio.h>

struct tree_t {
    int value;
    struct tree_t *left_ptr;
    struct tree_t *right_ptr;
};

tree_t* search(tree_t* t, int key) {
    while (t != NULL) {
        if (t->value == key) {
            return t;
        }
        if (t->value > key) {
            t = t->right_ptr;
        } else {
            t = t->left_ptr;
        }
    }
    return NULL;
}
```

L3:
```
cmpl %esi, (%rdi)
je L4
jle L6
movq 8(%rdi), %rdi
jmp L12
```

L6:
```
movq 16(%rdi), %rdi
```

L12:
```
testq %rdi, %rdi
jne L3
```

- Baseline
  - Same with and without -fno-in-conversion flag!
Another CMOV Example (Part II)

- gcc –Os –fno-if-conversion

```c
tree_t* search(tree_t* t, int key) {  
    while (t != NULL) {  
        if (t->value == key) {  
            return t;  
        }  
        tree_t* right = t->right_ptr;  
        tree_t* left = t->left_ptr;  
        if (t->value > key) {  
            t = right;  
        } else {  
            t = left;  
        }  
    }  
    return NULL;  
}
```

- Similar assembly as before (-fno-if-conversion)
  - Does reduce taken branches
Another CMOV Example (Part III)

• gcc -Os

tree_t* search(tree_t* t, int key) {
    while (t != NULL) {
        if (t->value == key) {
            return t;
        }
        tree_t* right = t->right_ptr;
        tree_t* left = t->left_ptr;
        if (t->value > key) {
            t = right;
        } else {
            t = left;
        }
    }
    return NULL;
}

• Now, with –fif-conversion (enabled by default)
  • Uses CMOV to avoid branch misprediction

L3:
   cmpl %esi, (%rdi)
   je L4
   movq 16(%rdi), %rax
   movq 8(%rdi), %rdi
   cmovle %rax, %rdi

L22:
   testq %rdi, %rdi
   jne L3
Predication Performance

- **Cost/benefit analysis**
  - Benefit: predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (fewer branches to track)
  - Cost: extra instructions (fetched, but not actually executed)

- **As branch predictors are highly accurate...**
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - Or even hurt!
  - But can help:
    - Deeper pipelines, hard-to-predict branches, and few added insn

- Thus, prediction is useful, but not a panacea
Readings

- Textbook (MA:FSPTCM)
  - Sections 3.1, 3.2 (but not “Sidebar” in 3.2), 3.5.1
  - Sections 4.2, 4.3, 5.3.3