Chapter 3

Instruction-Level Parallelism and Its Exploitation
Introduction

- Pipelining became a universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications
Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches
Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)

- Challenges:
  - Data dependency
    - Instruction \( j \) is data dependent on instruction \( i \) if
      - Instruction \( i \) produces a result that may be used by instruction \( j \)
      - Instruction \( j \) is data dependent on instruction \( k \) and instruction \( k \)
        is data dependent on instruction \( i \)

- Dependent instructions cannot be executed simultaneously
Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect
Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - Antidependence: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - Output dependence: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved

- To resolve, use renaming techniques
Other Factors

- **Data Hazards**
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)

- **Control Dependence**
  - Ordering of instruction $i$ with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch
Examples

- **Example 1:**
  DADDU R1,R2,R3
  BEQZ R4,L
  DSUBU R1,R1,R6
  L: ...
  OR R7,R1,R8

- **Example 2:**
  DADDU R1,R2,R3
  BEQZ R12,skip
  DSUBU R4,R5,R6
  DADDU R5,R4,R9
  skip:
  OR R7,R8,R9

- OR instruction dependent on DADDU and DSUBU

- Assume R4 isn’t used after skip
  - Possible to move DSUBU before the branch
Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction

- Example:
  for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Stalls

Loop:  

L.D   F0,0(R1)
stall
ADD.D F4,F0,F2
stall
stall
S.D F4,0(R1)
DADDUI R1,R1,#-8
stall (assume integer load latency is 1)
BNE R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline Scheduling

Scheduled code:
Loop:  L.D  F0,0(R1)
       DADDUI R1,R1,#-8
       ADD.D F4,F0,F2
       stall
       stall
       S.D F4,8(R1)
       BNE R1,R2,Loop

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>
Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

Loop:

```
L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1) ;drop DADDUI & BNE
L.D F6,-8(R1)
ADD.D F8,F6,F2
S.D F8,-8(R1) ;drop DADDUI & BNE
L.D F10,-16(R1)
ADD.D F12,F10,F2
S.D F12,-16(R1) ;drop DADDUI & BNE
L.D F14,-24(R1)
ADD.D F16,F14,F2
S.D F16,-24(R1)
DADDUI R1,R1,#-32
BNE R1,R2,Loop
```

- note: number of live registers vs. original loop
Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

Loop:
L.D F0,0(R1)
L.D F6,-8(R1)
L.D F10,-16(R1)
L.D F14,-24(R1)
ADD.D F4,F0,F2
ADD.D F8,F6,F2
ADD.D F12,F10,F2
ADD.D F16,F14,F2
S.D F4,0(R1)
S.D F8,-8(R1)
DADDUI R1,R1,#-32
S.D F12,16(R1)
S.D F16,8(R1)
BNE R1,R2,Loop
Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = n
  - Goal: make k copies of the loop body
  - Generate pair of loops:
    - First executes $n \mod k$ times
    - Second executes $n / k$ times
    - “Strip mining”
Branch Prediction

- **Basic 2-bit predictor:**
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction

- **Correlating predictor:**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding \( n \) branches

- **Local predictor:**
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last \( n \) occurrences of this branch

- **Tournament predictor:**
  - Combine correlating predictor with local predictor
Branch Prediction Performance

Branch predictor performance
Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow

- Advantages:
  - Compiler doesn’t need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time

- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions
Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion

- Creates the possibility for WAR and WAW hazards

- Tomasulo’s Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards
Register Renaming

- Example:

  DIV.D F0,F2,F4
  ADD.D F6,F0,F8
  S.D F6,0(R1)
  SUB.D F8,F10,F14
  MUL.D F6,F10,F8

  + name dependence with F6

  antidependence

Branch Prediction
Register Renaming

- Example:

  DIV.D F0,F2,F4
  ADD.D S,F0,F8
  S.D S,0(R1)
  SUB.D T,F10,F14
  MUL.D F6,F10,T

- Now only RAW hazards remain, which can be strictly ordered
Register Renaming

- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values
  - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
  - Pending instructions designate the RS to which they will send their output
    - Result values broadcast on a result bus, called the common data bus (CDB)
  - Only the last output updates the register file
  - As instructions are issued, the register specifiers are renamed with the reservation station
  - May be more reservation stations than registers
Tomasulo’s Algorithm

- Load and store buffers
  - Contain data and addresses, act like reservation stations

- Top-level design:
Tomasulo’s Algorithm

- Three Steps:
  - Issue
    - Get next instruction from FIFO queue
    - If available RS, issue the instruction to the RS with operand values if available
    - If operand values not available, stall the instruction
  - Execute
    - When operand becomes available, store it in any reservation stations waiting for it
    - When all operands are ready, issue the instruction
    - Loads and store maintained in program order through effective address
    - No instruction allowed to initiate execution until all branches that proceed it in program order have completed
  - Write result
    - Write result on CDB into reservation stations and store buffers
      - (Stores must wait until address and value are received)
## Example

### Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F6,32(R2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>L.D F2,44(R3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MUL.D F0,F2,F4</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB.D F8,F2,F6</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.D F10,F0,F6</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F6,F8,F2</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Reservation stations

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load1</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load2</td>
<td>Yes</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44 + Regs[R3]</td>
</tr>
<tr>
<td>Add1</td>
<td>Yes</td>
<td>SUB</td>
<td></td>
<td></td>
<td>Mem[32 + Regs[R2]]</td>
<td>Load2</td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>Yes</td>
<td>ADD</td>
<td></td>
<td></td>
<td>Add1</td>
<td></td>
<td>Load2</td>
</tr>
<tr>
<td>Add3</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mul1</td>
<td>Yes</td>
<td>MUL</td>
<td></td>
<td></td>
<td>Regs[F4]</td>
<td></td>
<td>Load2</td>
</tr>
<tr>
<td>Mul2</td>
<td>Yes</td>
<td>DIV</td>
<td></td>
<td></td>
<td>Mem[32 + Regs[R2]]</td>
<td>Mul1</td>
<td></td>
</tr>
</tbody>
</table>

### Register status

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F2</th>
<th>F4</th>
<th>F6</th>
<th>F8</th>
<th>F10</th>
<th>F12</th>
<th>...</th>
<th>F30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qi</td>
<td>Mul1</td>
<td>Load2</td>
<td>Add2</td>
<td>Add1</td>
<td>Mul2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution
Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit

- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?

- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit
Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared

- Exceptions:
  - Not recognized until it is ready to commit
Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors
### Multiple Issue

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution, but no speculation</td>
<td>None at the present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with speculation</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Primarily software</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler (often implicitly)</td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td>Primarily static</td>
<td>Primarily software</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>


VLIW Processors

- Package multiple operations into one instruction

- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references

- Must be enough parallelism in code to fill the available slots
VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility
Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation

- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches

- Issue logic can become bottleneck
Overview of Design
Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. on FP, one integer, one load, one store

- Examine all the dependencies among the instructions in the bundle

- If dependencies exist in bundle, encode them in reservation stations

- Also need multiple completion/commit
Example

Loop:  LD R2,0(R1)  ;R2=array element
      DADDIU R2,R2,#1  ;increment R2
      SD R2,0(R1)  ;store result
      DADDIU R1,R1,#8  ;increment pointer
      BNE R2,R3,LOOP  ;branch if not last element
### Example (No Speculation)

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock cycle number</th>
<th>Executes at clock cycle number</th>
<th>Memory access at clock cycle number</th>
<th>Write CDB at clock cycle number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>6</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#8</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>Execute directly</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>11</td>
<td>12</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#8</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>17</td>
<td>18</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>15</td>
<td>19</td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#8</td>
<td>8</td>
<td>14</td>
<td>15</td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Iteration number</th>
<th>Instructions</th>
<th>Issues at clock number</th>
<th>Executes at clock number</th>
<th>Read access at clock number</th>
<th>Write CDB at clock number</th>
<th>Commits at clock number</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td></td>
<td>6</td>
<td>7</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>7</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#8</td>
<td>2</td>
<td>3</td>
<td></td>
<td>4</td>
<td>8</td>
<td>Commit in order</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>8</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LD R2,0(R1)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>No execute delay</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>8</td>
<td></td>
<td>9</td>
<td>10</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>10</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#8</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7</td>
<td>11</td>
<td>Commit in order</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td>11</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LD R2,0(R1)</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>Earliest possible</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>11</td>
<td></td>
<td>12</td>
<td>13</td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SD R2,0(R1)</td>
<td>8</td>
<td>9</td>
<td></td>
<td>13</td>
<td>13</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#8</td>
<td>8</td>
<td>9</td>
<td></td>
<td>10</td>
<td>14</td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>13</td>
<td></td>
<td></td>
<td>14</td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>
Branch-Target Buffer

- Need high instruction bandwidth!
  - Branch-Target buffers
    - Next PC prediction buffer, indexed by current PC
Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”
Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites  
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack
Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines
Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order

- Simplifies commit:
  - Record that mapping between architectural register and physical register is no longer speculative
  - Free up physical register used to hold older value
  - In other words: SWAP physical registers on commit

- Physical register de-allocation is more difficult
Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (fixed number?)
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary
How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)

- Speculating through multiple branches
  - Complicates speculation recovery
  - No processor can resolve multiple branches per cycle
Energy Efficiency

- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance

- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors
  - Similar idea--address aliasing prediction--is used on some processors