Instruction-Level Parallelism and Its Exploitation
(Part III)

ECE 154B
Dmitri Strukov
Dealing With Control Hazards

• Simplest solution to stall pipeline until branch is resolved and target address is calculated
  – would severely limit dynamical scheduling
  – Loop unrolling might help somewhat
Solution 2: Delayed Branches

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect after that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay

- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper
A is the best choice, fills delay slot and reduces IC
In B and C, the sub instruction may need to be copied, increasing IC
In B and C, must be okay to execute sub when branch fails
Solution 3: Static Prediction

- Branch condition prediction only makes sense if delay for calculation condition is longer than that of target address
- Predict branch is always taken or always not taken
  - Compiler can help by favoring each case
  - Need mechanism to back off (flush pipeline when prediction is wrong)
- Even better solution is via profiling
  - Run program to get branch statistics
  - Explicitly specify direction in instruction
  - Branches are typically bimodal
Dynamic Branch Prediction

• Change the outcome of prediction in the runtime

• Basic 1 and 2-bit predictors:
  – For each branch:
    • Predict taken or not taken
    • If the prediction is wrong two consecutive times, change prediction

• Correlating or two-level 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
Basic Dynamic Branch Prediction

- **Branch History Table (BHT):** Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time (T-Taken, N)
  - No full address check (saves HW, but may be wrong)

- Problem: in a loop, 1-bit BHT will cause 2 mispredictions:
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - E.g., only 80% accuracy if 10 iterations per loop on average
2-bit Branch Prediction - Scheme 1

- Better Solution: 2-bit scheme:

(Jim Smith, 1981)
Branch History Table (BHT)

- BHT is a table of “Predictors”
  - 2-bit, saturating counters indexed by PC address of Branch
- In Fetch phase of branch:
  - Predictor from BHT used to make prediction
- When branch completes:
  - Update corresponding Predictor
Another Solution: 2-bit scheme where change prediction (in either direction) only if get misprediction twice:

Both schemes achieve 80-95% accuracy with only a small difference in behavior

Lee & A. Smith, IEEE Computer, Jan 1984
Branch Correlations

B1: if (x)
    ...
B2: if (y)
    ...
    z=x&&y
B3: if (z)
    ...

• B3 can be predicted with 100% accuracy based on the outcomes of B1 and B2
• Basic scheme cannot use this information
Correlating or Two-Level Branches

Idea: taken/not taken of recently executed branches is related to behavior of present branch (as well as the history of that branch behavior)
  - Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

• (2,2) predictor: 2-bit global, 2-bit local
  
Branch address (4 bits)

2-bits per branch local predictors

2-bit recent global branch history (01 = not taken then taken)
Accuracy of Different Schemes

4096 Entries 2-bit BHT
Unlimited Entries 2-bit BHT
1024 Entries (2,2) BHT
Global vs Local History Predictors

Global:
BHR: A shift register for global history
- Shift in latest result in each cycle
- provides global context

Local:
BHT: keeps track of local history
- select entry based on PC bits & shift in latest result in each cycle
Tournament Predictors

- Motivation for correlating branch predictors: 2-bit local predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use two predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Hopes to select right predictor for right branch (or right context of branch)
Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit is 0 => ith prior branch not taken;
  - ith bit is 1 => ith prior branch taken;
Tournament Predictor in Alpha 21264

- **Local predictor** consists of a 2-level predictor:
  - **Top level** a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted
  - **Next level** Selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction
- **Total size**: \(4K \times 2 + 4K \times 2 + 1K \times 10 + 1K \times 3 = 29K \text{ bits!} \quad (\sim 180K \text{ transistors})\)
% of predictions from local predictor in Tournament Prediction Scheme

- nasa7: 98%
- matrix300: 100%
- tomcatv: 94%
- doduc: 90%
- spice: 55%
- fpppp: 76%
- gcc: 72%
- espresso: 63%
- eqntott: 37%
- li: 69%
Accuracy of Branch Prediction

- Profile: branch profile from last execution (static in that is encoded in instruction, but profile)

Fig 3.40
Accuracy v. Size (SPEC89)

Conditional branch misprediction rate vs. Total predictor size (Kbits)

- Local - 2 bit counters
- Correlating - (2,2) scheme
- Tournament
- BHR: \{G, P\}: \{Global history, Per-address history\}
- PHT: \{g, p, s\}: \{Global, Per-address, Set\}
  - g: use the BHR output as the address into the PHT
  - p: combine the BHR output with some bits from the PC
  - s: use an arbitrary hashing function for PHT addressing
- 9 combinations: GAg, GAp, GAs, PAg, PAp, PAs, SAg, SAP and SAs

<table>
<thead>
<tr>
<th>predictor</th>
<th>l1_size</th>
<th>hist_size</th>
<th>l2_size</th>
<th>xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAg</td>
<td>1</td>
<td>W</td>
<td>$2^W$</td>
<td>0</td>
</tr>
<tr>
<td>GAp</td>
<td>1</td>
<td>W</td>
<td>$&gt;2^W$</td>
<td>0</td>
</tr>
<tr>
<td>PAg</td>
<td>N</td>
<td>W</td>
<td>$2^W$</td>
<td>0</td>
</tr>
<tr>
<td>PAp</td>
<td>N</td>
<td>W</td>
<td>$2^{N+W}$</td>
<td>0</td>
</tr>
<tr>
<td>gshare</td>
<td>1</td>
<td>W</td>
<td>$2^W$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Branch predictor parameters
Need Address at the Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch used as index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can’t use wrong branch address

<table>
<thead>
<tr>
<th>PC of instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Branch PC Predicted PC

=?

Yes: instruction is branch; use predicted PC as next PC (if predict Taken)

No: branch not predicted; proceed normally (PC+4)

Prediction state bits
Branch Target “Cache”

- Branch Target cache - Only predicted taken branches
- “Cache” - Content Addressable Memory (CAM) or Associative Memory (see figure)
- Use a big Branch History Table & a small Branch Target Cache

Branch PC

Yes: predicted taken branch found

No: not found

Prediction state bits (optional)
Steps with Branch target Buffer

for the 5-stage MIPS

Send PC to memory and branch-target buffer

Entry found in branch-target buffer?

Yes

Place predicted target address in PC

No

Place PC+4 in PC

Is instruction a taken branch?

Yes

Taken Branch?

No

Normal instruction execution

Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer

Enter branch instruction address and next PC into branch-target buffer. Kill fetched instruction. Restart fetch at other target.

Branch correctly predicted; continue execution with no stalls.

+ Prediction history table should be updated for all three cases
Branch Prediction for Function Returns

• Problem: Hard to predict target for returns (15% of all branches) because of indirection (60% accuracy)

• In most languages, function calls are fully nested
  – If you call A() → B() → C() → D()
  – Your return targets are PCc → PCb → PCA → PCmain

• Solution: Return Address Stack (RAS)
  – A FILO structure for capturing function return addresses
  – Operation: push on function call, pop on function return
  – 16-entry RAS is enough for the typical depth call trees
  – RAS mispredictions:
    • Overflows
    • Long jumps
Return Address Stack

16-entry → few percent of mispredictions
Branch Folding

Supply actual instruction instead of instruction address

- or 0 delay branch (or branch folding)
Multiple Issue

• 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</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>(static)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</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>(dynamic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>(speculative)</td>
<td></td>
<td></td>
<td>speculation</td>
<td></td>
<td></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</td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(often implicitly)</td>
<td></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</td>
<td>Itanium</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>the compiler</td>
<td></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
**Example for VLIW**

**Loop:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>Load array element F0 from memory</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>Add scalar F4 to F0 and store in F2</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>Store result F4 to memory</td>
</tr>
<tr>
<td>DADDUI R1,R1,#-8</td>
<td>Decrement pointer by 8 bytes</td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td>Branch if R1 != R2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer operation/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td></td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,-8(R1)</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td>S.D F16,-24(R1)</td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,16(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
</tr>
</tbody>
</table>

**Figure 3.16** VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled.
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
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
Instruction after branch can be issued but on hold until branch is resolved
For simplicity branches are issued in separate bundles

<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>
<tr>
<td>Iteration number</td>
<td>Instructions</td>
<td>Issues at clock number</td>
<td>Executes at clock number</td>
<td>Read access at clock number</td>
<td>Write CDB at clock number</td>
<td>Commits at clock number</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>------------------------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<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>
</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>
</tr>
<tr>
<td>1</td>
<td>SD R2,0(R1)</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>7</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>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>8</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>
</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>
</tr>
<tr>
<td>2</td>
<td>SD R2,0(R1)</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td>10</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>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td>11</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>
</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>
</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>
</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>
</tr>
<tr>
<td>3</td>
<td>BNE R2,R3,LOOP</td>
<td>9</td>
<td>13</td>
<td></td>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>
Limitations of ILP (perfect processor)

- Perfect cache
- Perfect branch and jump prediction
- Infinite register renaming
- Perfect memory address alias analysis
Limitations of ILP (more realistic but still ambitious)

- 64 int + 64 fp registers for renaming
- Tournament predictor
- Perfect memory address alias analysis
Motivation for Multithreaded Architectures

Processors not executing code at their hardware potential

- late 70’s: performance lost to memory latency
- 90’s: performance not in line with the increasingly complex parallel hardware
  - instruction issue bandwidth is increasing
  - number of functional units is increasing
  - instructions can execute out-of-order execution & complete in-order
  - still, processor utilization was decreasing & instruction throughput not increasing in proportion to the issue width

Major cause is the lack of instruction-level parallelism in a single executing thread

Therefore the solution has to be more general than building a smarter cache or a more accurate branch predictor
Multithreaded Processors

Multithreaded processors can increase the pool of independent instructions & consequently address multiple causes of processor stalling

- holds processor state for more than one thread of execution
  - registers
  - PC
  - each thread’s state is a hardware context
- execute the instruction stream from multiple threads without software context switching
- utilize thread-level parallelism (TLP) to compensate for a lack in ILP

3 styles: Coarse grain, fine grain, and simultaneous multithreading
CPU Support for Fine Grain MT

http://www.cs.manchester.ac.uk/ugt/2010/COMP25212/
Fine-Grain Multithreading

- Switch CPU threads on every cycle with minimal overhead
- Multithreading helps hiding stalls

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.cs.manchester.ac.uk/ugt/2010/COMP25212/
## Coarse Grain Multithreading

- Alternatively, if 1 CPU thread stalled, issue *every* clock from alternate thread

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>M-MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst d</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simultaneous Multi-Threading

“permit different threads to occupy the same pipeline stage at the same time”

- This makes most sense with superscalar issue

http://www.cs.manchester.ac.uk/ugt/2010/COMP25212/
Simultaneous MultiThreading

- Let’s look simply at instruction issue:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Inst a</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst b</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst M</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst N</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst c</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst P</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst Q</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst d</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst e</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inst R</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.cs.manchester.ac.uk/ugt/2010/COMP25212/
Multithreading

Traditional multithreaded processors hardware switch to a different context to avoid processor stalls

1. **coarse-grain** multithreading
   - switch on a long-latency operation (e.g., L2 cache miss)
   - another thread executes while the miss is handled
   - modest increase in instruction throughput with potentially no slowdown to the thread with the miss
     - doesn’t hide latency of short-latency operations
     - doesn’t add much to an already long stall
     - need to fill the pipeline on a switch
     - no switch if no long-latency operations
   - HEP, IBM RS64 III

2. **fine-grain** multithreading
   - can switch to a different thread each cycle (usually round robin)
   - hides latencies of all kinds
   - larger increase in instruction throughput but slows down the execution of each thread
   - Cray (Tera) MTA, Sun Niagara

3. **simultaneous multithreading (SMT)**
   - issues multiple instructions from multiple threads each cycle
   - no hardware context switching
   - huge boost in instruction throughput with less degradation to individual threads
Multithreading Example

[Diagram showing issue slots for Thread A, Thread B, Thread C, and Thread D over time. The diagram also compares different scheduling methods: Coarse MT, Fine MT, and SMT.]
Exceptions

• Exceptions are events that are difficult or impossible to manage in hardware alone
• Exceptions are usually handled by jumping into a service (software) routine
• Examples: I/O device request, page fault, divide by zero, memory protection violation (seg fault), hardware failure, etc.
Once an Exception Occurs, How Does the Processor Proceed?

- Non-pipelined: don’t fetch from PC; save state; fetch from interrupt vector table
- Pipelined: depends on the exception
  - Precise Interrupt: Must stop all instruction “after the exception” (squash)
  - Save state after last instruction before exception completes (PC, regs)
  - Fetch from interrupt vector table
  - Imprecise interrupts for out-of-order processors
    - Two modes of operations
    - Precise in speculative out of order processor exceptions are not recognize until instruction is ready to commit
Acknowledgements

Some of the slides contain material developed and copyrighted by I. Watson (U Manchester), by Csaba Moritz (U Amherst) and D. Culler (UCB) and instructor material for the textbook