Arclab Recitation
Agenda

- O3 Recap
- Gem5 & Konata Demo
Recap - Branch Prediction

● Problem:
  ○ Branch instructions gets “resolved” only after Execute stage.
  ○ Assuming no branch prediction, subsequent instructions in the pipeline have to be abandoned if the condition is taken.

● Solution:
  ○ Speculates outcome of conditional instructions.
  ○ If correctly predicted, we can avoid restarting pipeline and wasting processor cycles.
Dealing With Branches
Recap - Branch Prediction

- Two prediction questions
  - Direction? - taken or not-taken
  - Target? - next PC to fetch
- Implemented with BTB (Branch Target Buffer)
  - Input: current PC + some history of past branches
  - Output: next PC
- Detailed explanation in lecture slides.
2-bit logic branch predictor

```c
int globalPredictor = 0;

// lookup: Generate prediction for a branch
bool lookup() {
    return globalPredictor / 2;
}

// update: Update branch predictor value after branch outcome.
void update(bool taken) {
    globalPredictor = ((globalPredictor << 1) & 3) | taken;
}
```
2-bit logic branch predictor

<table>
<thead>
<tr>
<th>Current State</th>
<th>Taken</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/N</td>
<td>N</td>
<td>00/N</td>
</tr>
<tr>
<td>00/N</td>
<td>T</td>
<td>01/N</td>
</tr>
<tr>
<td>01/N</td>
<td>N</td>
<td>10/T</td>
</tr>
<tr>
<td>01/N</td>
<td>T</td>
<td>11/T</td>
</tr>
<tr>
<td>10/T</td>
<td>N</td>
<td>00/N</td>
</tr>
<tr>
<td>10/T</td>
<td>T</td>
<td>01/N</td>
</tr>
<tr>
<td>11/T</td>
<td>N</td>
<td>10/T</td>
</tr>
<tr>
<td>11/T</td>
<td>T</td>
<td>11/T</td>
</tr>
</tbody>
</table>
Recap - Register Renaming

- Anti (WAR) and output (WAW) dependencies are false dependencies. They can be eliminated if we have unlimited # of registers.
  - We do not actually have unlimited # of registers, but this is still possible because processors usually have much more physical registers than architectural registers.
  - E.g. the “Alpha” ISA has 32 integer and 32 floating-point architectural registers but the “Alpha 21264” microprocessor which implements this ISA, has 80 integer and 72 floating-point physical registers.
Recap - Register Renaming (example)

- Architectural registers: r1, r2, r3
- Physical registers: p1, p2, p3, p4, p5, p6, p7
- Identify all WAW’s and WAR’s, notice how renaming eliminates them

<table>
<thead>
<tr>
<th>MapTable</th>
<th>FreeList</th>
<th>Raw insns</th>
<th>Renamed insns</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>p1, p4</td>
<td>add r2,r3\rightarrow r1</td>
<td>add p2,p3\rightarrow p4</td>
</tr>
<tr>
<td>r2</td>
<td>p2, p2</td>
<td>sub r2,r1\rightarrow r3</td>
<td>sub p2,p4\rightarrow p5</td>
</tr>
<tr>
<td>r3</td>
<td>p3, p4</td>
<td>mul r2,r3\rightarrow r1</td>
<td>mul p2,p5\rightarrow p6</td>
</tr>
<tr>
<td>p1</td>
<td>p4, p5,p6,p7</td>
<td>div r1,r3\rightarrow r2</td>
<td>div p6,p5\rightarrow p7</td>
</tr>
</tbody>
</table>
Recap - Dynamic Scheduling

- Allow instructions to enter Execute stage out-of-order
- Program correctness (e.g. RAW dependencies) is ensured by Reservation Stations and Reorder Buffer.
- Understanding how they work is crucial for Arch Lab, Part C.
Cycle: 001

```c
// Implement DAXPY here
for(i = 0; i < N; i++)
{
    Y[i] = (alpha * X[i] + Y[i]);
}
```

```assembly
movl $0, -4(%rbp)
jmp .L2

.L3:
ia1  movsd X,(%rax,8), %xmm1
ia2  movsd alpha(%rip), %xmm0
ia3  mulsd %xmm1, %xmm0
ia4  movsd Y,(%rax,8), %xmm1
ia5  addsd %xmm1, %xmm0
ia6  movsd %xmm0, Y,(%rax,8)
ia7  addl $1, -4(%rbp)

.L2:
ia8  cmpl $2055, -4(%rbp)
ia9  jle .L3
```

Four instructions (ia1, ia2, ia3, ia4) have been dispatched to RS with four corresponding entries allocated in ROB. ia1 and ia2 have been issued into LS FU. Next cycle, ia1 will finish and broadcast its result using tag "%rob1" to ia3.
Cycle: 002

```
// Implement DAXPY here
for(i = 0; i < N; i++)
{
    Y[i] = (alpha * X[i] + Y[i]);
}
```

movl $0, -4(%rbp)
jmp .L2

.L3:
ia1  movsd X, (%rax, 8), %xmm1
ia2  movsd alpha, (%rip), %xmm0
ia3  mulsd %xmm1, %xmm0
ia4  movsd Y, (%rax, 8), %xmm1
ia5  addsd %xmm1, %xmm0
ia6  movsd %xmm0, Y, (%rax, 8)
ia7  addl $1, -4(%rbp)

.L2:
ia8  cmpl $2055, -4(%rbp)
ia9  jle .L3

ia1 completes and is next to leave ROB.
Its RS entry has been reallocated to ia5.
ia4 is issued into LS FU.
Next cycle, ia2 will broadcast its results using tag "%rob2" to ia3.
Cycle: 003

```plaintext
movl $0, -4(%rbp)
jmp .L2
.L3:
ia1  movsd X, (%rax, 8), %xmm1
iaux  movsd alpha, (%rip), %xmm0
ia3  mulsd %xmm1, %xmm0
ia4  movsd Y, (%rax, 8), %xmm1
ia5  addsd %xmm1, %xmm0
ia6  movsd %xmm0, Y, (%rax, 8)
ia7  addl $1, -4(%rbp)
.L2:
ia8  cmpl $2055, -4(%rbp)
ia9  jle .L3
```
Cycle: 004

```
// Implement DAXPY here
for (i = 0; i < N; i++)
{
    Y[i] = (alpha * X[i] + Y[i]);
}
```

movl $0, -4(%rbp)
jmp .L2

.L3:
ia1: movsd X, (%rax, 8), %xmm1
nia2: movsd alpha, (%rip), %xmm0
nia3: mulsd %xmm1, %xmm0
nia4: movsd Y, (%rax, 8), %xmm1
nia5: addsd %xmm1, %xmm0
nia6: movsd %xmm0, Y, (%rax, 8)
tia7: addl $1, -4(%rbp)

.L2:
tia8: cmpl $2055, -4(%rbp)
tia9: jle .L3

ia4 finishes but must wait for ia3 before it can leave ROB.
ia7 has been dispatched to RS and allocated entry in ROB, and is issued.
Next cycle, ia3 will forward result to ia5 using the tag “%rob3”.

---

Trace Table:

<table>
<thead>
<tr>
<th>B</th>
<th>I</th>
<th>F</th>
<th>IA</th>
<th>RR</th>
<th>S</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Instructions:
- ia7 finishes and is issued.
- ia3 enters ROB.
- ia5 enters ROB.
- ia4 enters ROB.
- ia8 is dispatched to RS.

---

Trace Table:

<table>
<thead>
<tr>
<th>ROB</th>
<th>tag</th>
<th>TP</th>
<th>HP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ia7</td>
<td>ia6</td>
<td>ia5</td>
<td>ia4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Important observations @ c4

- Register renaming removes WAR dependency between ia3 and ia4 on %xmm1, and thus allow out-of-order execution
  - ia4 finishes before ia3 once WAR is removed @ c4
- RS entries are freed earlier than ROB entries
  - RS entries are freed as soon as execution finishes (e.g. ia1@c1, ia2@c2, & ia4@c3), but ROB entry only gets freed if it is at head of the ROB (e.g. ia1@c3, ia2@c4).
  - This should help in choosing optimal RS/ROB sizes and provide analysis (last part) in part C of Arch Lab.
Recap - Load Bypassing & Forwarding

- Implemented with Load & Store Queues (LSQ)
- Detailed example in lecture slides.

![Diagram showing Load Bypassing and Load Forwarding](image-url)
Gem5 Demo

```bash
ssh <andrewid>@shark.ics.cs.cmu.edu
cd ~/private
wget <Link to be provided>
tar xvf recX.tar.gz
cd recX
make
```
Gem5 Demo

- Gem5 can help us to collect performance metrics and analyze performance bottlenecks.
- Consider the daxpy example we went through earlier, what happens if we tweak ROB and IQ size?

- Let’s start with IQ=4 and ROB=4
- `./run_gem5.py --directory=./out/daxpy44 --cmd=./daxpy --IQ=4 --ROB=4`
- Look into stats.txt, where do you think the bottleneck is? How can you tell?
- So many metrics, which should we care the most?
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch_cpus.rename.serializeStallCycles</td>
<td>252</td>
</tr>
<tr>
<td>switch_cpus.rename.RunCycles</td>
<td>2105</td>
</tr>
<tr>
<td>switch_cpus.rename.UnblockCycles</td>
<td>24508</td>
</tr>
<tr>
<td>switch_cpus.rename.RenamedInsts</td>
<td>36745</td>
</tr>
<tr>
<td>switch_cpus.rename.ROBFullEvents</td>
<td>22371</td>
</tr>
<tr>
<td>switch_cpus.rename.IQFullEvents</td>
<td>18</td>
</tr>
<tr>
<td>switch_cpus.rename.RenamedOperands</td>
<td>42112</td>
</tr>
<tr>
<td>switch_cpus.rename.RenameLookups</td>
<td>88238</td>
</tr>
<tr>
<td>switch_cpus.rename.int rename lookups</td>
<td>37053</td>
</tr>
<tr>
<td>switch_cpus.rename.fp rename lookups</td>
<td>22899</td>
</tr>
<tr>
<td>switch_cpus.rename.CommittedMaps</td>
<td>41288</td>
</tr>
<tr>
<td>switch_cpus.itb.rdMisses</td>
<td>0</td>
</tr>
<tr>
<td>switch_cpus.itb.wrMisses</td>
<td>51</td>
</tr>
<tr>
<td>system.cpu.workload.numSyscalls</td>
<td>2</td>
</tr>
<tr>
<td>switch_cpus.pwrStateResidencyTicks::OFF</td>
<td>36491400</td>
</tr>
<tr>
<td>switch_cpus.numCycles</td>
<td>121953</td>
</tr>
<tr>
<td>switch_cpus.numWorkItemsStarted</td>
<td>0</td>
</tr>
<tr>
<td>switch_cpus.numWorkItemsCompleted</td>
<td>0</td>
</tr>
<tr>
<td>switch_cpus.fetch.icacheStallCycles</td>
<td>12687</td>
</tr>
<tr>
<td>switch_cpus.fetch.Insts</td>
<td>24827</td>
</tr>
<tr>
<td>switch_cpus.fetch.Branches</td>
<td>4064</td>
</tr>
<tr>
<td>switch_cpus.fetch.predictedBranches</td>
<td>2220</td>
</tr>
</tbody>
</table>

- **switch_cpus.rename.serializeStallCycles**: Count of cycles rename stalled for serializing instructions
- **switch_cpus.rename.RunCycles**: Number of cycles rename is running
- **switch_cpus.rename.UnblockCycles**: Number of cycles rename is unblocking
- **switch_cpus.rename.RenamedInsts**: Number of instructions processed by rename
- **switch_cpus.rename.ROBFullEvents**: Number of times rename has blocked due to ROB full
- **switch_cpus.rename.IQFullEvents**: Number of times rename has blocked due to IQ full
- **switch_cpus.rename.RenamedOperands**: Number of destination operands rename has renamed
- **switch_cpus.rename.RenameLookups**: Number of register rename lookups that rename has made
- **switch_cpus.rename.int rename lookups**: Number of integer rename lookups
- **switch_cpus.rename.fp rename lookups**: Number of floating rename lookups
- **switch_cpus.rename.CommittedMaps**: Number of HB maps that are committed
- **switch_cpus.itb.rdMisses**: TLB misses on read requests
- **switch_cpus.itb.wrMisses**: TLB misses on write requests
- **system.cpu.workload.numSyscalls**: Number of system calls
- **switch_cpus.pwrStateResidencyTicks::OFF**: Cumulative time (in ticks) in various power states
- **switch_cpus.numCycles**: Number of cpu cycles simulated
- **switch_cpus.numWorkItemsStarted**: Number of work items this cpu started
- **switch_cpus.numWorkItemsCompleted**: Number of work items this cpu completed
- **switch_cpus.fetch.icacheStallCycles**: Number of cycles fetch is stalled on an Icache miss
- **switch_cpus.fetch.Insts**: Number of instructions fetch has processed
- **switch_cpus.fetch.Branches**: Number of branches that fetch encountered
- **switch_cpus.fetch.predictedBranches**: Number of branches that fetch has predicted taken
Gem5 Demo

- Based on stats.txt, which buffer should we enlarge? ROB or IQ?

- Try with (IQ, ROB) = (8, 4) and (4, 8), which one actually helps with performance?
  - ./run_gem5.py --directory=./out/daxpy48 --cmd=./daxpy --IQ=4 --ROB=8
  - ./run_gem5.py --directory=./out/daxpy84 --cmd=./daxpy --IQ=8 --ROB=4

- Now try with (IQ, ROB) = (8, 8). Will this configuration improve performance further? Why or why not?
  - ./run_gem5.py --directory=./out/daxpy88 --cmd=./daxpy --IQ=8 --ROB=8
Konata Demo

- Konata is a visualization tool that helps with reasoning traces generated by Gem5
- Download at https://github.com/shioyadan/Konata/releases

- Compare trace of (IQ, ROB) = (4, 8) and (4, 4), what difference in patterns can we notice by using Konata?
  - Start looking from around line 200
Konata Demo (IQ, ROB) = (4, 4)
Konata Demo (IQ, ROB) = (4, 8)
Konata Demo - Observations

- F corresponds to IQ, Rn corresponds to ROB
- Instructions stays longer in ROB in (4, 4), also causes later instructions to “waste” cycles in IQ
- Validates that ROB size is the bottleneck in daxpy

- Typically the closer \#IQFullEvent with \#ROBFullEvent, the better
- Represented in Konata, we should prefer to see a smooth “exit slope”