BASELINE 2-WAY INORDER SUPERSCALAR PARALLEL PROCESSOR

- Processors studied so far are fundamentally limited to CPI > 1
- Superscalar processors enable CPI < 1 (i.e., IPC > 1) by executing multiple instructions in parallel
- Can have Tomasulo and out-of-order superscalar processors. Start with exploring inorder.

Simplify discussion of cycle count.

Fetch 2 instructions at once.

Regen needs of read ports.

A pipe: int ops, branches
B pipe: int ops, mem ops

Register file needs 2 write ports.

Issue logic "issues" instruction to appropriate execution pipe.

More abstract way to show pipe stages.
**Issue Logic**

\[ \text{CPI} = 0.5, \text{IPC} = 2 \]

- Multiple instructions in stages F, D, W acceptable because we have a superscalar processor with duplicated hardware to avoid structural hazard.

- Instructions "swapped" from natural fetch position to appropriate execution pipe.

- Stall due to structural hazard on F pipeline. Can only process one memory op per cycle. (ie cannot have 2 units in D0 at same time)

- Stall due to RAW within fetch pipeline.

- Bypassing still must stall for one cycle.

- Order is important.

Here we do not have a RAW dependency. We do not need to stall fourth instruction to get result from 3rd unit. And we definitely do not stall 3rd because its some matches best of 4th.

Is there a WAR hazard?
Fetch Logic

CAREFULLY MANAGE FETCH ALIGNMENT

<table>
<thead>
<tr>
<th>CRC</th>
<th>ADDR</th>
<th>WITH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000</td>
<td>op A</td>
</tr>
<tr>
<td>0</td>
<td>0x004</td>
<td>op D</td>
</tr>
<tr>
<td>1</td>
<td>0x008</td>
<td>op C</td>
</tr>
<tr>
<td>1</td>
<td>0x00C</td>
<td>J 0x100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x100</td>
<td>op D</td>
</tr>
<tr>
<td>2</td>
<td>0x104</td>
<td>J 0x204</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x204</td>
<td>op E</td>
</tr>
<tr>
<td>3</td>
<td>0x208</td>
<td>J 0x30C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0x30C</td>
<td>op F</td>
</tr>
<tr>
<td>4</td>
<td>0x310</td>
<td>op G</td>
</tr>
<tr>
<td>5</td>
<td>0x314</td>
<td>op H</td>
</tr>
</tbody>
</table>

DEALING WITH MISALIGNED MULTIMUX.
FETCH WITHIN A CACHE LINE IS
TRICKY, MISALIGNED ACROSS CACHE
LINES VERY TRICKY. (CONSIDER DEP TRACKING)

SIMPLIFIES IMPLEMENTATION TO ONLY
FETCH ALIGNED TICS, POSSIBLY
THROWING AWAY FIRST INSTR.

<table>
<thead>
<tr>
<th>ADDR</th>
<th>WITH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>2 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>3 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x300</td>
<td>4 4</td>
</tr>
<tr>
<td>0x310</td>
<td>5 5</td>
</tr>
</tbody>
</table>

PIPELINE DIAGRAM
OF NO ALIGNMENT CONSTRAINTS
Precise Exceptions

Just as we need to carefully track program order to prevent data dependencies, similarly need to be careful with exceptions.

LW      F D 00 01 W — Assume address exception
SYSCALL F D 00 01 W — Also causes exception

Commit point, need to handle LW exception first even though SySCALL is in A pipe.
Bypassing

Bypassing network can quickly become very complex.
This and other reasons motivate us to add a dedicated issue stage.

D = Decode, possibly resolve some structural hazards
I = Reg file read, bypassing issue instruction
to appropriate functional unit

<table>
<thead>
<tr>
<th>op</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FDI</td>
<td>FDI</td>
<td>FDE</td>
<td>FDE</td>
</tr>
</tbody>
</table>

Branch Delay Latency

<table>
<thead>
<tr>
<th>op</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FDI</td>
<td>FDI</td>
<td>FDI</td>
<td>FDI</td>
</tr>
</tbody>
</table>

3 cycle penalty
Squash 7 instructions!
Branch Prediction

Essential in modern processors to avoid mitigate branch delay penalties.

2 Key types of prediction

1. Predict Branch Outcome

2. Predict Branch/Jump Target Address

Pipe Pipeline

F D I X M W

→ Know Branch Outcome

→ Know Target for JR, JALR

→ Know Target for Branches, J, JAL

- Focus first on SW Techniques then HW Techniques
- Focus first on predicting branch outcome, then branch/jump target
SW BPRD: Branch Delay Slots

\[
\begin{align*}
\text{BR} & : \text{FDxMW} \\
\text{opA} & : \text{FD} \\
\text{opB} & : \text{FDxMW} \\
\text{target} & : \text{FDxMW}
\end{align*}
\]

If delay slots must squash fall through instr if taken branch.

\[
\begin{align*}
\text{BR} & : \text{FDxMW} \\
\text{opA} & : \text{FD} \\
\text{opB} & : \text{FDxMW} \\
\text{target} & : \text{FDxMW}
\end{align*}
\]

If delay slots not useful work in slots.Instr in delay slot always executed regardless of branch cond.

Exposes too much about microarch, less popular now

SW BPRD: Static SW prediction

Extend ISA so that compiler can tell microarchitecture if branch is likely to be taken or not

\[
\begin{align*}
\text{BR} & : \text{FDxMW} \\
\text{opA} & : \text{FD} \\
\text{target} & : \text{FDxMW} \\
\text{opB} & : \text{FDxMW} \\
\text{opC} & : \text{FD} \\
\text{opD} & : \text{FDxMW}
\end{align*}
\]

As soon as branch is decoded use hint to redirect control flow

TAKEN/NOT TAKEN "HINT" FROM COMPILER

What if hint is wrong?

\[
\begin{align*}
\text{BR} & : \text{FDxMW} \\
\text{opA} & : \text{FDxMW} \\
\text{target} & : \text{FD} \\
\text{opD} & : \text{FDxMW}
\end{align*}
\]

Check hint in X stage if wrong squash target + redirect control flow

Notice we do not need to recheck opA if we wait one cycle to squash it.
Static SW prediction can work because we can exploit static structure in the program.

For example, most branch instructions are taken so use not to specify taken for these branches.

**SW OPRED:** Prediction

Not really "prediction." Idea is to turn control flow into data flow completely eliminating control hazard.

\[
\begin{align*}
\text{movn rd, rs, r+} & \quad \text{if } (R[r+]) = 0 \quad R[rd] \leftarrow R[rs] \\
\text{movz rd, rs, r+} & \quad \text{if } (R[r+]) = 0 \quad R[rd] \leftarrow R[rs]
\end{align*}
\]

**Pseudo Code**

<table>
<thead>
<tr>
<th>If (a &lt; b)</th>
<th>(x = a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Else</td>
<td>(x = b)</td>
</tr>
</tbody>
</table>

**Full OPREDication**

- Almost all instructions can be executed under predicate
- Instruction becomes NOP if predicate is false

\[
\begin{align*}
\text{if } (a < b) & \quad \text{slt} p, r1, r2, r3 \\
\text{else} & \quad \text{nop}
\end{align*}
\]

**Instruction**

\[
\begin{array}{c}
\text{op A} \\
\text{op B} \\
\text{else} \\
\text{op C} \\
\text{op D}
\end{array}
\]

**Special predicate flag**
HW Branch: Fixed Prediction

1. Always Predict NOT-Taken
   - what we have been assuming
   - simple to implement
   - know fall through PC in F
   - poor accuracy, especially on very important backwards branches in loops.

2. Always Predict TAKEN
   - difficult to implement because don't know target until K
   - could still achieve branch delay latency in PARC 14 one
   - poor accuracy, especially on if, then, else

3. Backward Branch TAKEN, Forward Branch NOT TAKEN
   - similar to (2) in terms of target
   - better accuracy

```
loop:
  lw   r1, 0(r2)
  lw   r3, 0(r4)
  slt  r5, r1, r3
  beq  r5, r1
  move r6, r1
  j     
  li:
  move r6, r3
  sw   r6, 0(r7)
  addw r2, r2, 4
  addw r4, r4, 4
  addw r7, r7, 4
  addw r8, r8, -1
  b+4  r8, loop
```

Forward Branch on avg 50% taken

Backward Branch on avg 90% taken
HW Brand: Exploiting Temporal Correlation

Exploit structure in program: the way a branch resolves may be a good indication of the way it will resolve the next time it is executed (temporal correlation).

1-bit Saturating Counter

Remember condition of last branch execution + predict it goes same way

<table>
<thead>
<tr>
<th>Situation</th>
<th>Predicted</th>
<th>Actual</th>
<th>Mispredict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>NT</td>
<td>Y</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>NT</td>
<td>Y</td>
</tr>
</tbody>
</table>

Always 2 mispredicts. Loops are very common. Exploiting temporal correlation works very well.
### 2-bit Saturating Counter

<table>
<thead>
<tr>
<th>Heaton</th>
<th>Predicted</th>
<th>Actual</th>
<th>Misprediction?</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
<td>Strong NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>Y</td>
<td>Weak NT</td>
</tr>
<tr>
<td>J</td>
<td>T</td>
<td>T</td>
<td></td>
<td>Weak T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>NT</td>
<td></td>
<td>Strong T</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td></td>
<td>Strong T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td></td>
<td>Strong T</td>
</tr>
<tr>
<td>J</td>
<td>T</td>
<td>NT</td>
<td>Y</td>
<td>Weak T</td>
</tr>
</tbody>
</table>

**What if START STATE is Strong Taken?**

Only 1 misprediction per loop.
Other 2-bit FSM Branch Predictors

Jump directly to strong from weak

Biased towards predicting branches as taken

See Fig. 5.8 in Sued & Lipasti for other alternatives
Implementing FSMs with Branch History Table

FSM State for specific PC

Two PC's can "alias" to the same entry in BHT. Similar to a cache conflict except we can't really tell when a conflict happens.

Storing the PC as a tag is too expensive.

\[ \text{Aliasage} = \frac{\text{Mispredictions}}{\text{True Positives}} \]

\[ \text{Aliasage of larger but on can alias + associativity} \text{ (still only use a few THT as "tag" in associative structure.)} \]

Most BHT are direct mapped because with only 2-3 bits per entry cheap to use large but

\[ \# \text{ bits/entry but} \times 80-90\% \text{ prediction accuracy} \]

How do we continue to improve prediction accuracy? Exploit more sophisticated temporal correlation.

More Complicated Temporal Correlation

Often a branch exhibits more complicated pattern than just "always taken" or "always not taken". Could develop a more complicated FSM. But these patterns vary per branch.

We want per branch customized FSMs.

```c
void convolve( int b[], int A[], int size ) {
    for ( int i = 2; i < size-2; i++ )
        for ( int j = 0; j < 5; j++ )
            A[i-j] = b[i] * COEFF[j];
}
```

Can we predict that every fifth dynamic instance of the backwards loop branch will be not taken?
2-Level BUS to exploit temporal correlation

When a branch is taken or not taken we shift in either a one (taken) or zero (not taken) into the least significant bit of the corresponding BUSR.

The BUSR captures the temporal pattern for that branch.

We use BUSR to index into the PHT. A PHT has an entry per branch, but a PHT has an entry per branch pattern.

The PHT says for a given pattern over the past n executions of a branch, should I take or not take the next execution of this branch?

So in previous example pattern for inner loop backwards branch will be

```
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ...
```

With 2-level BUS and for n > 4 we can learn this pattern and perfectly predict this branch.

Problem: missing in the BUSR

Solution: make BUSR larger or increase associativity.
**Problem:** Multiple branches with same history might need different projections. In other words, aliasing in one put can move accuracy.

**Solution:** Add multiple puts, use bits from PC to choose which put to use.
HW Branch: Exploiting Spatial Correlation

The way one branch is resolved may be a good indicator of the way a later (different) branch will resolve.

```
if (x < 7)
  y++
  if (x < 5)
    z++
    s1+ r2, r1, 7
    be2 r2, L1     branch 1
    a0010 r3, r3, 1
    L1:
    s1+ r2, r1, 5
    be2 r2, L2     branch 1
    a0010 r4, r4, 1
    L2:
```

If branch 1 is taken (i.e. x > 7)
then branch 1 is always taken (i.e. x must be > 5)

So whether branch 1 is taken or not taken can be used to predict if we should take branch 1.

For above example, Bushwill capture history - so we will know when first branch is taken, and that value (s) of
the Bush will point to an entry in the PUT that predicts taken.
As before, multiple Pts can help avoid aliasing in Pkt.
GENERALIZED TWO-LEVEL BULTS

Combined approach to exploit both complex temporal correlation and spatial correlation.

Difference from discussion on complex temporal correlation is that we purposely choose a smaller \( m \) to cause aliasing in the BULT.

Since this aliasing allows us to capture spatial correlation.

(Choose higher order \( m \) bits)

<table>
<thead>
<tr>
<th>ONE BUS</th>
<th>ONE BUS</th>
<th>ONE BUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 0 )</td>
<td>( 0 &lt; k &lt; 30 )</td>
<td>( k = 30 )</td>
</tr>
<tr>
<td>( m = 0 )</td>
<td>( G_A )</td>
<td>( G_A )</td>
</tr>
<tr>
<td>( 0.2m &lt; 30 )</td>
<td>( P_A )</td>
<td>( P_A )</td>
</tr>
<tr>
<td>( m = 50 )</td>
<td>( S_A )</td>
<td>( S_A )</td>
</tr>
</tbody>
</table>

97% accuracy
Q-SHARE

ISO MORPHIC TO PREVIOUS FIGURE

Q-SELECT

Instead of concatenating
16-bit PC with
T宽, helps avoid
aliasing in the put/out
more effectively.
TOURNAMENT PREDICTORS

DIFFERENT PREDICTORS ARE BETTER AT PREDICTING DIFFERENT TYPES OF BRANCHES

- one-level 2-bit counting counter
- two-level guess
- loops
- irregular code

Branch predictor selector table
- predicts which branch predictor we should use
HW BtB: Predicting Target Address

Even with best possible prediction of Branch outcome, still need to wait for target address to be determined.

Branch Target Buffer

PC

\[ \begin{array}{c|c|c}
V & PC & Target State \\
\hline
/0 & /0 & /0 \\
\end{array} \]

Hit? \quad \text{New PC = Predicted Target PC} \\
\text{if hit and predicted taken}

Can put BTB in Fetch stage:
- Predicting if PC points to a Branch
- Predicting Target of Branch
- Predicting if Branch is taken.

Sometimes \( b=0 \), if hit then assume predict taken
COMBINE BTB AND BUT

BTB is much more expensive than but, but BTB earlier in pipeline can accelerate JR

COMBINE BTB & few entries with but & many entries

RETURN ADDRESS STACK PREDICTOR

BTB only works for JR function call returns if always call function from same place (not realistic)

STACK PREDICTOR
  - Push target address on stack for JAL/JALR
  - Pop off target address for JR to predict target

Move stack predictor into fetch and predict which PC's are JR.

Use tournament predictor to choose between BTB and stack predictor.