Handling Hazards

- **Structural hazards**
  - Stalling: pipeline interlock
  - Code scheduling

- **Data hazards**
  - Stalling: pipeline interlock
  - Forwarding
  - Load delay
    - Stalling: pipeline interlock
    - Code scheduling: fill the load delay slot

- **Control Hazards**
  - Early branch resolution
  - Stalling: flushing the pipeline
  - Delayed branch
  - Predict non taken (or taken)
  - Static branch prediction
  - Dynamic branch prediction
Exceptions and Pipelining

- Recoverable exceptions is a challenge (e.g. pagefault, arithmetic exception)
  - Instructions before must commit
  - Instruction after must be killed
  - Instructions causing exception must be re-executed with same operands
  - Exceptions may overlap in pipeline, or even occur out-of-order
- Handling of exception delayed until Instruction reaches last stage.
Multicycle Operations

- Floating point operations take multiple cycles in EXE
- Assume a system with 1 int ALU, 1 FP/int multiplier, 1 FP adder, 1 FP/int divider
Multicycle Operations

- **Instruction latency**: cycles to wait for the result of an instruction
  - Usually the number of cycles for the execution pipeline minus 1
  - e.g. 0 cycles for integer ALU since no wait is necessary

- **Instruction initiation interval**: time to wait to issue another
  instruction of the same type
  - Not equal to number of cycles, if multicycle operation is pipelined or partially pipelined

- **Examples**:
  - Integer ALU:
    1 EXE cycle → latency = 0; initiation interval = 1
  - FP add, fully pipelined:
    4 EXE cycles → latency = 3; initiation interval = 1
  - FP divide, not pipelined:
    25 EXE cycles → latency = 24; initiation interval = 25
Multicycle Operations: Handling Hazards

- Structural hazards can occur when functional unit not fully pipelined (initiation interval > 1) → need to add interlocking
- RAW hazards become longer
- Possibly more than one register write per cycle → either add ports to register file or treat conflict as a hazard and stall
- Possible hazards between integer and FP instructions → use separate register files
- WAW hazards are possible → stall second instruction or prevent first instruction from writing
Loop Example

\[
\text{for (i=1000; i>0; i--)} \\
\quad x[i] = x[i] + s
\]

- Straightforward code and schedule:
  - Assume \textbf{F2} contains the value of \( s \)
  - Load latency equals 1
  - FP ALU latency 3 to another FP ALU and 2 to a store

\[
\begin{align*}
\text{loop: } & \text{L.D F0,0(R1) ;F0=\text{array element}} \\
& \text{stall ;add depends on ld} \\
& \text{ADD.D F4,F0,F2 ;main computation} \\
& \text{stall ;st depends on add} \\
& \text{stall} \\
& \text{S.D F4,0(R1) ;store result} \\
& \text{DADDUI R1,R1,-8 ;decrement index} \\
& \text{stall ;bne depends on add} \\
& \text{BNE R1,R2,loop ;next iteration} \\
& \text{stall ;branch delay slot}
\end{align*}
\]
Iteration Scheduling

- Execution time of straightforward code: 10 cycles/element
- Smart compiler (or human 😊) schedule:
  - Immediate offset of store was changed after reordering
  - Execution time of scheduled code:
    - 6 cycles/element → Speedup=1.7
  - Of the 6 cycles, 3 are for actual computation (l.d, add.d, s.d), 2 are loop overhead (addi, bne), and 1 is a stall
Next topic:
handling control hazards through branch prediction
Static Branch Prediction

- Compiler determines whether branch is likely to be taken or likely to be not taken.
  - How?

When is a branch likely to be taken?

When is a branch likely to be NOT taken?

```c
for(i=0;i<1000;i++)
{
    x = a[i];
    if (x==-1)
    {
        break;
    }
    process(x,i);
}
```
Static Branch Prediction

- Compiler determines whether branch is likely to be taken or likely to be not taken.

- Decision is based on analysis or profile information
  - 90% of backward-going branches are taken
  - 50% of forward-going branches are not taken
  - BTFN: “backwards taken, forwards not-taken”
  - Used in ARC 600 and ARM 11

- Decision is encoded in the branch instructions themselves
  - Uses 1 bit: 0 => not likely to branch, 1=> likely to branch

- Prediction may be wrong!
  - Must kill instructions in the pipeline when a bad decision is made
  - Speculatively issued instructions must not change processor state
Recall: Control Hazards

- When a branch is executed, PC is not affected until the branch instruction reaches the MEM stage.
- By this time 3 instructions have been fetched from the fall-through path.

Kill instructions in EX, DEC and IF as they move forwards
Dynamic Branch Prediction

- Monitor branch behavior and learn
  - Key assumption: past behavior indicative of future behavior
- Predict the present (current branch) using learned history
- Identify individual branches by their PC or dynamic branch history
- Predict:
  - Outcome: taken or not taken
  - Target: address of instruction to branch to
- Check actual outcome and update the history
- Squash incorrectly fetched instructions
Simplest dynamic predictor: 1-bit Prediction

- 1 bit indicating Taken (1) or Not Taken (0)

- Branch prediction buffers:
  - Match branch PC during IF or ID stages

  ![Branch PC Table]

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x135c8</td>
<td>0</td>
</tr>
<tr>
<td>0x147e0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Incurs at least 2 mis-predictions per loop

Problem: “unstable” behavior

```c
for(i=0; i<1000; i++)
{
    x = a[i];
    if (x==-1) {
        break;
    }
    process(x,i);
}
```
2-bit Branch Prediction

- **Idea: add hysteresis**
  - Prevent spurious events from affecting the most likely branch outcome

- **2-bit saturating counter:**
  - 00: do not take
  - 01: do not take
  - 10: take
  - 11: take

<table>
<thead>
<tr>
<th>Branch PC</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x135c8</td>
<td>10</td>
</tr>
<tr>
<td>0x147e0</td>
<td>11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

```c
for(i=0;i<1000;i++)
{
    x = a[i];
    if (x==-1){
        break;
    }
    process(x,i);
}
```
2-bit Branch Prediction

- Predictor states:
  - Learns biased branches
  - N-bit predictor:
    - Increment on Taken outcome and decrement on Not Taken outcome
    - If counter>$(2^n-1)/2$ then take, otherwise do not take
    - Takes longer to learn, but sticks longer to the prediction
Example of 2-bit Branch Prediction

- Nested loop:  
  
  Loop1: ...
  ...
  Loop2: ...
  bne r1,r0,loop2
  ...
  bne r2,r0,loop1

- 1\textsuperscript{st} outer loop execution:
  - 00 → predict not taken; actually taken → update to 01 (misprediction)
  - 01 → predict not taken; actually taken → update to 10 (misprediction)
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - ...
  - 11 → predict taken; actually not taken → update to 10 (misprediction)
Example Continued

- 2\textsuperscript{nd} outer loop execution onwards:
  - 10 → predict taken; actually taken → update to 11
  - 11 → predict taken; actually taken
  - ...
    - 11 → predict taken; actually not taken → update to 10 (misprediction)

- In practice misprediction rates for 2-bit predictors with 4096 entries in the buffer range from 1\% to 18\% (higher for integer applications than for fp applications)

- Bottom-line: 2-bit branch predictors work very well for loop-intensive applications
  - n-bit predictors (n>2) are not much better
  - Larger buffer sizes do not perform much better
Correlating Predictors

- 1- and 2-bit predictors exploit most recent history of the current branch
- Realization: branches are correlated!

```c
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    ...
}
```

If both branches are taken, the last branch definitely not taken

```c
char s1 = "Bob"
...
if (s1 != NULL)
    reverse_str(s1);
    
reverse_str(char *s) {
    if (s1 == NULL)
        return;
    ...

    s1 definitely not Null in this calling context
```
Correlating Predictors

- 1- and 2-bit predictors exploit most recent history of the current branch

- Realization: branches are correlated!

```c
if (a == 2)
    a = 0;
if (b == 2)
    b = 0;
if (a != b) {
    ...
}

char s1 = "Bob"
...  
if (s1 != NULL)
    reverse_str(s1);

reverse_str(char *s) {
    if (s1 == NULL)
        return;
    ... 

Idea: exploit recent history of other branches in prediction
Two-Level (or Correlating) Predictor

- Prediction depends on the context of the branch
- Context: history (T/NT) of the last N branches
  - First level of the predictor
  - Implemented as a shift register
- Prediction: 2-bit saturating counters
  - Indexed with the “global” history
  - Second level of the predictor

Global History Register GHR)

Pattern History Table (PHT)
Dynamic Predictor Performance Comparison

2-level correlating predictor with a 15-bit global history

2-level predictor improves accuracy by >4%
Does 4% accuracy improvement matter?

- Assume branches resolve in stage 10
  - Reasonable for a modern high-frequency processor
- 20% of instructions are branches
- Correctly-predicted branches have a 0-cycle penalty (CPI=1)
- 2-bit predictor: 92% accuracy
- 2-level predictor: 96% accuracy

2-bit predictor:
\[ \text{CPI} = 0.8 + 0.2 \times (10 \times 0.08 + 1 \times 0.92) = 1.114 \]

2-level predictor
\[ \text{CPI} = 0.8 + 0.2 \times (10 \times 0.04 + 1 \times 0.96) = 1.072 \]

Speedup(2-level over 2-bit): 4%
Branch Target Buffers (BTB)

- Branch predictors tell whether the branch will be taken or not, but they say nothing about the target of the branch.
- To resolve a branch early we need to know both the outcome and the target.
- Solution: store the likely target of the branch in a table (cache) indexed by the whole of the branch PC → BTB.
- Usually BTB is accessed in the IF stage and the branch predictor is accessed soon after that.
- (Aside: In the practical assignment, you will assume a perfect BTB, and a branch predictor that predicts in the fetch stage. Also, branches are resolved in the execute state.)
Complete Branch Prediction Logic

Source: Onur Mutlu, CMU
Branch Prediction with only BTB

IF stage

ID stage

EXE stage

Search BTB using instruction PC

Entry found in BTB?

No

Yes

Is this a taken branch?

No

Yes

Continue execution

Add branch to BTB

Use target in BTB

No

Yes

Taken branch?

No

Squash target; remove entry from BTB

Continue execution

Continue execution