Pipelined Datapath

Reading

- Sections 4.5 – 4.10
- Practice Problems: 1, 3, 8, 12
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Pipeline Speedup

• If all stages are balanced
  ➢ i.e., all take the same time

\[
\text{Inter} - \text{instruction} - \text{gap}_{\text{pipelined}} = \frac{\text{Inter} - \text{instruction} - \text{gap}_{\text{nonpipelined}}}{\text{number} - \text{of} - \text{stages}}
\]

• If not balanced, speedup is less
• Speedup due to increased throughput
  ➢ Latency (time for each instruction) does not decrease

---

Basic Idea

All instructions are 32-bits
Few & regular instruction formats
Alignment of memory operands

---
Pipelining

- What makes it easy
  - All instructions are the same length
  - Simple instruction formats
  - Memory operands appear only in loads and stores

- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

- What really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
Graphically Representing Pipelines

- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths
Structural Hazard

Need to separate instruction and data memory

IF for Load, Store, ...

(11)

(12)
ID for Load, Store, ...

EX for Load
MEM for Load

Wrong register number

WB for Load
Corrected Datapath for Load

EX for Store
MEM for Store

WB for Store
Pipelining Example

Note what is happening in the register file

 Pipeline stage execution time

Pipelined Control (Simplified)
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along like data
Datapath with Control

IF: xxxx  ID: xxxx  EX: add $14, $8, $9  MEM: or $13, ...
WB: and $12...

Datapath with Control

IF: xxxx  ID: xxxx  EX: xxxx  MEM: add $14, ...
WB: or $13...
Datapath with Control

Data Hazards (4.7)

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

```
<table>
<thead>
<tr>
<th>Time</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
<th>1200</th>
<th>1400</th>
<th>1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0, $t0, $t1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $t2, $s0, $t3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Problem with starting next instruction before first is finished

- dependencies that “go backward in time” are data hazards

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>register $2$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Program execution order

1. sub $2$, $1$, $3$
2. and $12$, $2$, $5$
3. or $13$, $6$, $2$
4. add $14$, $2$, $2$
5. sw $15$, 100($2$)

Software Solution

- Have compiler guarantee no hazards
- Where do we insert the “nops”? 

`sub $2$, $1$, $3$
and $12$, $2$, $5$
or $13$, $6$, $2$
add $14$, $2$, $2$
sw $15$, 100($2$)`

Problem: this really slows us down!
A Better Solution

- Consider this sequence:
  
  sub $2, $1, $3  
  and $12, $2, $5  
  or $13, $6, $2  
  add $14, $2, $2  
  sw $15, 100($2)

- We can resolve hazards with forwarding
  - How do we detect when to forward?

---

Dependencies & Forwarding

Program execution order (in instructions):

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)

Do not wait for results to be written to the register file – find them in the pipeline → forward to ALU

---

(37)
Forwarding (from EX/MEM)

Forwarding (from MEM/WB)
Forwarding (operand selection)

Forwarding (operand propagation)

Combinational Logic!
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

But only if forwarding instruction will write to a register!
- EX/MEM.RegWrite, MEM/WB.RegWrite

And only if Rd for that instruction is not $zero
- EX/MEM.RegisterRd ≠ 0,
  MEM/WB.RegisterRd ≠ 0
Forwarding Paths

b. With forwarding

Forwarding Conditions

• EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

• MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Double Data Hazard

- Consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4

- Both hazards occur
  - Want to use the most recent

- Revise MEM hazard condition
  - Only forward if EX hazard condition isn’t true

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
  - Checking precedence of EX hazard
Correct execution is about managing dependencies
- Producer-consumer
- Structural (using the same hardware component)

We will come across other types of dependencies later!
Load-Use Data Hazard

Need to stall for one cycle

Forwarding
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - (ID/EX.RegisterRt = IF/ID.RegisterRs) or
    - (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E; C = B + F;$

\[
\begin{array}{c}
\text{lw } \$t1, 0(\$t0) \\
\text{lw } \$t2, 4(\$t0) \\
\text{add } \$t3, \$t1, \$t2 \\
\text{sw } \$t3, 12(\$t0) \\
\text{lw } \$t4, 8(\$t0) \\
\text{add } \$t3, \$t1, \$t2 \\
\text{sw } \$t3, 12(\$t0) \\
\text{lw } \$t5, 8(\$t0) \\
\text{add } \$t5, \$t1, \$t4 \\
\text{sw } \$t5, 16(\$t0)
\end{array}
\]

11 cycles 13 cycles
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB perform a **nop** (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for **lw**
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

Program execution order (in instructions):
- `lw $2, 20($1)`
- and becomes nop
- `and $4, $2, $5`
- or $6, $2, $6
- add $9, $4, $2

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

hw $2, 20($1)
and becomes nop
and $4, $2, $5 stalled in ID
or $8, $2, $6 stalled in IF
add $9, $4, $2

Or, more accurately...

Datapath with Hazard Detection

ALUSrc mux is missing!
Control Hazards (4.8)

- Branch instruction determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline cannot always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and determine the branch condition

Branch Hazards

- If branch outcome determined in MEM

![Diagram of branch hazards](image)

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
  - Add IF.Flush signal to squash IF/ID register

- Example: branch taken
  
  ```
  36:  sub  $10, $4, $8
  40:  beq  $1, $3, 72
  44:  and  $12, $2, $5
  48:  or   $13, $2, $6
  52:  add  $14, $4, $2
  56:  slt  $15, $6, $7
  ...
  72:  lw   $4, 50($7)
  ```

Example: Branch Taken

![Diagram of processor architecture showing branch processing stages and signals](image)
Example: Branch Taken

Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} preceding ALU instruction

\[
\begin{align*}
\text{add } & \$1, \$2, \$3 & \text{ IF } & \text{ ID } & \text{ EX } & \text{ MEM } & \text{ WB } \\
\text{add } & \$4, \$5, \$6 & \text{ IF } & \text{ ID } & \text{ EX } & \text{ MEM } & \text{ WB } \\
\ldots & & \text{ IF } & \text{ ID } & \text{ EX } & \text{ MEM } & \text{ WB } \\
\text{beq } & \$1, \$4, \text{ target} & \text{ IF } & \text{ ID } & \text{ EX } & \text{ MEM } & \text{ WB }
\end{align*}
\]

- Can resolve using forwarding
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2\textsuperscript{nd} preceding load instruction
  - Need 1 stall cycle

\begin{align*}
\text{lw} & \quad \text{\$1, addr} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{add} & \quad \text{\$4, \$5, \$6} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{beq} & \quad \text{stalled} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{beq} & \quad \text{\$1, \$4, target} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB}
\end{align*}

(67)

Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

\begin{align*}
\text{lw} & \quad \text{\$1, addr} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{beq} & \quad \text{stalled} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{beq} & \quad \text{stalled} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB} \\
\text{beq} & \quad \text{\$1, \$0, target} & \text{IF} & \text{ID} & \text{EX} & \text{MEM} & \text{WB}
\end{align*}

(68)
Delay Slot (MIPS)

- Expose pipeline
- Load and jump/branch entail a “delay slot”
- The instruction right after the jump or branch is executed before the jump/branch

```
jal   function A
add   $4, $5, $6 ; executed before jmp
lw    $12, 8($4) ; executed after return
```

- Jump/branch and the delay slot instruction are considered “indivisible”
- In the delay slot, the compiler needs to schedule
  - A useful instruction (either before the jmp, or after the jmp w/o side effect)
  - otherwise a NOP

Branch Prediction

- Longer pipelines cannot readily determine branch outcome early
  - Stall penalty becomes unacceptable

- Predict outcome of branch
  - Only stall if prediction is wrong

- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
MIPS with Predict Not Taken

Program execution order
(time in instructions)

Prediction correct

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

```
outer: ...

inner: ...

beq ..., ..., inner

beq ..., ..., outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-Bit Predictor: State Machine

- Only change prediction on two successive mispredictions

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Later in this course

ECE 6100

Instruction Level Parallelism (ILP)

Later in this course

AMD Bobcat

http://hothardware.com

Intel Sandy Bridge

bdti.com
Exceptions and Interrupts (4.9)

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently

- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, ...

- Interrupt
  - From an external I/O controller

- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)

- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)

- Save indication of the problem
  - In MIPS: Cause register
    - We’ll assume 1-bit
      - 0 for undefined opcode, 1 for overflow

- Jump to handler at 80000180
An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause

- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - ...: C000 0040

- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, ...
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  \[ \text{add } $1, $2, $1 \]
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler

- Similar to mispredicted branch
  - Use much of the same hardware
Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Re-fetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

- Exception on `add` in
  40  `sub  $11, $2, $4`
  44  `and  $12, $2, $5`
  48  `or  $13, $2, $6`
  4c  `add $1, $2, $1`
  50  `slt  $15, $6, $7`
  54  `lw  $16, 50($7)`

- Handler
  80000180  `sw  $25, 1000($0)`
  80000184  `sw  $26, 1004($0)`

(83)
Exception Example

hw $15, 50($7)  
alt $15, $6, $7  
add $1, $2, $1  
or $13, . . .  
and $12, . . .

Exception Example

sw $25, 1000($0)  
bubble (nop)  
bubble  
bubble  
or $13, . . .

Clock 6

Clock 7
Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once

- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions

- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)

- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion

- Simplifies hardware, but more complex handler software

- Not feasible for complex multiple-issue out-of-order pipelines
Performance

- How do we assess the impact of stall cycles?
- How close do we approach the ideal of one instruction per cycle execution time?
- Back to the CPI model!

Recall: Program Execution time

\[ \text{ExecutionTime} = \left( \sum_{i=1}^{n} C_i \times CPI_i \right) \times \text{cycle}_\text{time} \]

\[ \sim \text{Instruction}_\text{count} \times \text{CPI}_{\text{avg}} \times \text{clock}_\text{cycle}_\text{time} \]

\[ \text{CPI}_{\text{avg}} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left( \frac{\text{CPI}_i \times \text{Instruction Count}}{\text{Instruction Count}} \right) \]

Relative frequency

algorithms/compiler  architecture  technology
Assessing Performance

• Ideal CPI is increased by dependencies

• Performance impact on CPI can be assessed by computing the impact on a per instruction basis

\[ \text{Increase in CPI} = \text{Base CPI} + \text{Probability of event} \times \text{penalty for event} \]

- For example, an event may be a branch misprediction or the occurrence of a data hazard
- The probability is computed for the occurrence of the event on an instruction

• Examples: pipelined processors

Instruction-Level Parallelism (ILP) (4.10)

• Pipelining: executing multiple instructions in parallel

• To increase ILP
  - Deeper pipeline
    - Less work per stage \(\Rightarrow\) shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages \(\Rightarrow\) multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice
Multiple Issue

- **Static multiple issue**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- **Dynamic multiple issue**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

MIPS with Static Dual Issue

- **Two-issue packets**
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
MIPS with Static Dual Issue

Instruction Level Parallelism (ILP)

- Single (program) thread of execution
- Issue multiple instructions from the same instruction stream
- Average CPI<1
- Often called out of order (OOO) cores
Dynamically Scheduled CPU

- Instruction fetch and decode unit
  - In-order issue
  - Preserves dependencies

- Reservation station
- Integer
- Functional units
- Hold pending operands
- Out-of-order execute

- Reservation station
- . . .
- Floating point

- Reservation station
- Load-store
- Results also sent to any waiting reservation stations
- In-order commit

- Commit unit
  - Can supply operands for issued instructions

Reorders buffer for register writes

AMD Bulldozer

System Bus

- Instruction Fetch and PreDecode
- Instruction Queue
- Decode
- Rename/Alloc
- Reorder Buffer
  - Retirement Unit
- Schedulers
- L1 D-Cache and D-TLB

L2 Cache and Control

- Instruction Fetch and PreDecode
- Instruction Queue
- Decode
- Rename/Alloc
- Reorder Buffer
  - Retirement Unit
- Schedulers
- L1 D-Cache and D-TLB

(forum.beyond3d.com)
AMD Bobcat

Later in this course

ECE 6100

Instruction Level Parallelism (ILP)

Later in this course

ECE 6100

http://hothardware.com

The P4 Microarchitecture

From, ”The Microarchitecture of the Pentium 4 Processor 1,” G. Hinton et al, Intel Technology Journal Q1, 2001
Study Guide

• Given a code block, and initial register values (those that are accessed) be able to determine state of all pipeline registers at some future clock cycle.
• Determine the size of each pipeline register
• Track pipeline state in the case of forwarding and branches
• Compute the number of cycles to execute a code block
• Modify the datapath to include forwarding and hazard detection for branches (this is trickier and time consuming but well worth it)

Study Guide (cont.)

• Schedule code (manually) to improve performance, for example to eliminate hazards and fill delay slots
• Modify the data path to add new instructions such as \texttt{j}
• Modify the data path to accommodate a two cycle data memory access, i.e., the data memory itself is a two cycle pipeline
  ❖ Modify the forwarding and hazard control logic
• Given a code sequence, be able to compute the number of stall cycles
Study Guide (cont.)

- Track the state of the 2-bit branch predictor over a sequence of branches in a code segment, for example a for-loop
- Show the pipeline state before and after an exception has taken place.

Glossary

- Branch prediction
- Branch hazards
- Branch delay Control hazard
- Data hazard
- Delay slot
- Dynamic instruction issue
- Forwarding
- Imprecise exception
- Instruction scheduling
- Instruction level parallelism (ILP)
- Load-to-use hazard
- Pipeline bubbles
- Stall cycles
- Static instruction issue
- Structural hazard