Data path for load/store instructions of the form

lw $reg1, offset($reg2) 16 bits
sw $reg1, offset($reg2)

Address is $reg1 contents + offset (can be negative)
For control, see Prof. Gottlieb's lecture notes lectures 14-18 (first part)

http://cs.nyu.edu/courses/fall02/V22.0436-001/lectures/lectures.html

- Drawings come from P+H.
Pipelining

Chapter 6, P+H (just 6.1, 6.8, 6.9)

- Motivation -
  1. Break instruction execution into multiple stages (on MIPS: 5)
  2. As the hardware finishes a stage of an instruction, have it start on the same stage of the next instruction.

Pipeline Stages:

  - Instruction fetch
  - Instruction decode or register read
  - Execute operation or calculate address (ALU)
  - Memory access
  - Write result to register

So, ideally, pipeline looks like:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>ALU</th>
<th>MEM</th>
<th>REGW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>REGW</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>REGW</td>
</tr>
<tr>
<td></td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>REGW</td>
</tr>
</tbody>
</table>
```

Legend:

- IF: Instruction fetch
- ID: Instruction decode
- ALU: ALU execute
- MEM: Memory access
- REGW: Register write
Thus, instead of the total execution being:

\[ N \times CPI \]

where \( N \) = total number of instructions executed and \( CPI \) = cycles per instruction, the total pipelined execution time is (ideally)

\[ N + CPI - 1 \]

Typically, each stage in the pipeline is 1 cycle, so CPI is the number of stages.

The MIPS instruction set is designed for pipelining:

1. All instructions are the same size
   - only one instruction fetch cycle/decode stage

2. Register operands are always in the same place
   - this operand fetch can occur at the same time as instruction decoding

3. Most operations are simple and fit well into the 5 stage pipeline.
"Ideally," an instruction can start executing every cycle, but that isn't always possible.

Why?
- structural hazards
- control hazards
- data hazards

Structural Hazards: When two instructions in different stages of the pipeline require the same hardware resource.

Example: If a single memory is used (with a single port), then IF and Mem stages cannot occur simultaneously.

- See last instruction in previous drawing
- Other structural hazards include bus contention; contention for ALU
- Can be aided by good choice of ISA.

When a hazard occurs, an instruction may be delayed until the resource is free.

This results in a stage being "unfilled" for a cycle, causing a bubble.
Control Hazards

What instruction should follow a branch instruction?
- Don't know until the condition is computed (target address can be computed at same time).

Solutions:
1. Stall the pipeline
   - i.e. and don't perform IF for the next instruction until the branch has been resolved (i.e. to branch or not)
   - this inserts bubbles into the first stage

2. Perform branch prediction
   - guess which way the branch will go, and start IF for that instruction on time
   - if wrong, flush the incorrect instruction from the pipeline
   - it's important to guess right most of the time:
     - possibilities: always guess branch not taken

   use static/compile analysis
   - e.g. branch back usually at bottom of loop
   use branch prediction hardware
(solutions to control hazards continued)

3. Always execute the instruction following the branch, regardless of whether the branch is taken or not — "branch delay slot"

- Compiler or assembler makes sure the instruction following the branch is the appropriate function
- e.g. move an instruction from above the branch into the branch delay slot.

- not always possible (usually about 50% of the time)

Data Hazards:

What if the value required by one instruction in the pipeline is being computed by another instruction in the pipeline?

Example:

```
add $50, $51, $52
sub $54, $53, $30
```

Solutions:
1) Don't have the compiler schedule these instructions like this — can't always avoid.
2) Stall the second instruction until $50 has been written to.
(Solution to data hazards, continued)

3. Perform "forwarding" (also "bypassing")
   - write directly from one stage of the pipeline to another
   - bypassing the need to wait for the register write.

Example from before:

```
add $x0, $x1, $x2
sub $x4, $x3, $x0
```

![Pipeline Diagram]

You still have to be careful to ensure that the "source" stage executes before the "target" stage during bypassing.
- Still might need to stall.

Example

```
lw $x4, 100($x2)
add $x5, $x3, $x2
```

![Pipeline Diagram]

Compiler can help:

```
lw $x1, 100($x4)
lw $x2, 200($x3)
sw $x2, 100($x4)
sw $x1, 200($x3)
```

3. Interchange these two
   3. To remove data hazard.
The Memory Hierarchy

Chap. 7 in Pt.M

"Memory hierarchy" refers to several layers of memory.

- fast, small, cache (primary storage)
- main memory
- slow, large, disk

- hopefully, data and instructions that are about to be needed are already in fast memory (i.e., cache)
- if a needed datum or instruction is found in cache, it's called a "cache hit"
- otherwise, it's a "cache miss", and slower memory is accessed.

Notice analogy: cache hits/misses $\leftrightarrow$ page faults in V.M.

Typically, hierarchy contains 4 levels, but could be any number (in theory): L1, L2, main, disk

Purpose: Exploit temporal and spatial locality
Cache Organization - 3 major categories:
- Direct maged
- Fully associative
- Set associative

**Direct Mapped Cache**
- Simplest organization
- Word is placed in cache according to its address in main memory.
  - Since cache is smaller than main memory, many addresses in memory get mapped to same location in cache typically:
    - \( \text{Mem address mod num-words-in-cache} \)
  - If number of words in cache is a power of 2, then mapping is easy!

\[ \text{memory address} \]

\[ \frac{32 - 2}{\log_2(N)} = \log_2(N) \text{ bits} \]

\[ N = \text{num words in cache} \]

- Just look at bits 2 - \( \frac{(\log_2(N) + 1)}{2} \)

Example: 256 KB primary cache, \( \log_2(N) = 18 \div 2 = 16 \)
- Use bits 2 - 17 to index into cache.