32-bit RISC
Pipelining

• Improve performance by increasing instruction throughput
Pipelining

• What makes it easy
  – all instructions are the same length
  – just a few 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

• We’ll build a simple pipeline and look at these issues

• We’ll talk about modern processors and what really makes it hard:
  – exception handling
  – trying to improve performance with out-of-order execution, etc.
Basic Idea

- **What do we need to add to actually split the datapath into stages?**
Pipelined Datapath

Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?
Pipelined Execute

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
Instruction Fetch (Load)
Instruction Decode
Execution
Memory
Write Back
Pipeline control

• We have 5 stages. What needs to be controlled in each stage?
  – Instruction Fetch and PC Increment
  – Instruction Decode / Register Fetch
  – Execution
  – Memory Stage
  – Write Back
Pipeline Control

- Pass control signals along just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst ALU Op1 ALU Op0 ALU Src Branch Mem Read Mem Write Reg write Mem to Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>1 1 0 0 0 0 0 0 1 0 0 1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>0 0 0 1 0 1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>X 0 0 1 0 0 1 0 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>X 0 1 0 1 0 0 0 X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Datapath with Control
Dependencies

- 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>Value of register $2$:</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

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)`
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!
Forwarding

- Use temporary results, don’t wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding

<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>Value of register $2$ :</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
<td>–20</td>
</tr>
<tr>
<td>Value of EX/MEM :</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB :</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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)
Forwarding

Control

IF/ID

Instruction memory

PC

Instruction

Registers

Forwarding unit

ID/EX

EX/MEM

MEM/WB

Data memory

ALU

Mux

Mux

Mux

Mux

Mux

Forwarding unit

EX/MEM, RegisterRd

MEM/WB, RegisterRd

EX/MEM, RegisterRd

MEM/WB, RegisterRd

EX/MEM, RegisterRd

MEM/WB, RegisterRd

IF/ID, RegisterRs

IF/ID, RegisterRt

IF/ID, RegisterRt

IF/ID, RegisterRd
Load Word

• **Load word can still cause a hazard:**
  – an instruction tries to read a register following a load instruction that writes to the same register.

• Thus, we need a hazard detection unit to “stall” the load instruction
Stalling

• We can stall the pipeline by keeping an instruction in the same stage

```
Stalling

• We can stall the pipeline by keeping an instruction in the same stage
```

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC 1</td>
</tr>
</tbody>
</table>

lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7

Program execution order (in instructions)
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward
Branch Hazards

- When we decide to branch, other instructions are in the pipeline!

- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong
Flush Instructions
Pipelined Architecture
Improving Performance

• **Try and avoid stalls!** E.g., *reorder* these instructions:

  lw $t0, 0($t1)
  lw $t2, 4($t1)
  sw $t2, 0($t1)
  sw $t0, 4($t1)

• **Add a “branch delay slot”**
  – the next instruction after a branch is always executed
  – rely on compiler to “fill” the slot with something useful

• **Superscalar**: start more than one instruction in the same cycle
Branch delay slot

a. From before

    add $s1, $s2, $s3
    if $s2 = 0 then
    Delay slot

    Becomes

    if $s2 = 0 then
    add $s1, $s2, $s3

b. From target

    sub $t4, $t5, $t6
    ... 
    add $s1, $s2, $s3
    if $s1 = 0 then
    Delay slot

    Becomes

    add $s1, $s2, $s3
    if $s1 = 0 then
    sub $t4, $t5, $t6

c. From fall through

    add $s1, $s2, $s3
    if $s1 = 0 then
    Delay slot

    Becomes

    add $s1, $s2, $s3
    if $s1 = 0 then
    sub $t4, $t5, $t6
Instruction Level Parallelism
Instructon Level Parallelism

• High speed execution based on Instruction Level Parallelism (ILP): potential of short instruction sequences to execute in parallel

• High-speed microprocessors exploit ILP by:
  – pipelined execution: overlap instructions
  – superscalar execution: issue and execute multiple instructions per clock cycle
  – Out-of-order execution

• Memory accesses for high-speed microprocessor
  – Data Cache, possibly multiported, multiple levels
• Pipeline
• Superscalar
• Superpipeline
• Vector Processing
• VLIW
• Dynamic Scheduling
• Branch Prediction
• Cache
Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**
- Cycle 1: Load
- Cycle 2: Store

**Multiple Cycle Implementation:**
- Cycle 1: Load
- Cycle 2: Ifetch
- Cycle 3: Reg
- Cycle 4: Exec
- Cycle 5: Mem
- Cycle 6: Wr
- Cycle 7: Ifetch
- Cycle 8: Reg
- Cycle 9: Exec
- Cycle 10: Mem
- Cycle 11: Ifetch

**Pipeline Implementation:**
- Load: Ifetch, Reg, Exec, Mem, Wr
- Store: Ifetch, Reg, Exec, Mem, Wr
- R-type: Ifetch, Reg, Exec, Mem, Wr
Pipeline Property

- Pipeline does not help latency of single task, it helps throughput of entire workload
- Pipeline rate is limited by slowest pipeline stage
- Multiple tasks operate simultaneously
- Potential Speedup = No. of pipeline stages
  \[
  \text{Time per instruction on non-pipelined machine} \div \text{Number of pipe stages}
  \]
- Pipeline Hazards reduce the performance from the ideal speedup gained by pipelining
Superscalar

- Parallel execute different instructions
  - check dependencies (data or control)
- Superscalar execution with degree n=3

```
Cycle 0 1 2 3 4 5 6 7 8 9
```

ifetch decode execute writeback
Performance of superscalar

- **m-issue, N instructions, k stages**
- **The time required by the scalar base machine**
  \[ T(1,1) = N + k - 1 \] (base cycles)
- **Ideal execution time of m-issue superscalar machine**
  \[ T(m,1) = k + \frac{(N - m)}{m} \] (base cycles)
- **Ideal speed up**
  \[ S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{(N + k - 1)}{\left\{\frac{N}{m} + k - 1\right\}} \]
Superpipeline

- Superpipeline of degree n
- Cycle time is 1/n of base cycle
- Superpipelined execution with degree n= 3
Superscalared superpipeline

- Superpipelined superscalar of degree(3,3)

- Superpipelined superscalar of degree(m,n)
  - executes m instructions every cycle with a pipeline cycle 1/n of base cycle
Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

**SCALAR**

(1 operation)

\[
\begin{align*}
\text{add } r3, r1, r2
\end{align*}
\]

**VECTOR**

(N operations)

\[
\begin{align*}
\text{add.vv } v3, v1, v2
\end{align*}
\]
Properties of Vector Processors

- Each result independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate

- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency
  => no data caches required! (Do use instruction cache)

- Reduces branches and branch problems in pipelines

- Single vector instruction implies lots of work
  => fewer instruction fetches
VLIW (Very Long Instruction Word)

- Originated from parallel microcode
- Code compaction by compiler
Properties of VLIW

- Easy decoding
- Possibility of having low code density
- Different parallelism require different instruction sets
- Random parallelism
  - Regular parallelism when Vector and SIMD
Dynamic Scheduling

- **Hardware rearranges instruction execution to prevent stalls**
  - enables handling unknown dependencies (e.g., memory reference) and simplifies compiler
  - enables compiled code to run efficiently on different platforms
  - complicates hardware
  - complicates exception handling
    - imprecise exceptions
    - difficult to restart after interrupt

- **Dynamic Ordering**
  - Issue
  - Completion
Out-of-order Issue

- **Central Window**
  - Complex (de)allocation
  - Need capability of any type of instruction
  - Selects among a larger number of instructions

- **Reservation Station**
  - Partition instruction by functional unit
  - Simple, duplicated control logic
  - Larger number of entry at equivalent performance
Control Hazard

• **Modern microprocessor**
  – deeper and deeper pipeline
  – wider and wider execution

• **control hazard**
  – frequent bubble maker
  – significant performance factor

• **How to solve hazard?**
  – Loop counter
  – Conditional instruction
  – Branch prediction
Static Branch Prediction

- **Predict-taken**
  - MIPS-X

- **Predict-not-taken**
  - Motorola MC88000

- **Prediction bit in instruction**
  - compiler support
  - PowerPC, SPARC V9
Dynamic Prediction

• **Branch Prediction**
  – Branch address
  – Branch history
    • taken or not-taken about previous branches
    • two-level adaptive
  – Hybrid
    • correlated

• **Branch Target Buffer**
  – lookup cache with branch address
  – target address without computing
    • fetch target instruction without stall
Cache Principles

- The Dilemma: fast, large and not expensive memory
- Locality solves the dilemma
- Memory Hierarchy

- Design Consideration
  - Cache Size
  - Block Mapping
  - Replacement Strategy
  - Write Policy
  - Unified / Separated
  - Cache Coherency
Improve Cache Performance

1. Reduce the miss rate
2. Reduce the miss penalty
3. Reduce the time to hit in the cache.
Where to misses come from?

- **Classifying Misses: 3 Cs**
  - **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called \textit{cold start misses} or \textit{first reference misses}.
    \textit{(Misses in even an Infinite Cache)}
  - **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, \textit{capacity misses} will occur due to blocks being discarded and later retrieved.
    \textit{(Misses in Fully Associative Size X Cache)}
  - **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called \textit{collision misses} or \textit{interference misses}.
    \textit{(Misses in N-way Associative, Size X Cache)}

- **4th “C”:**
  - **Coherence** - Misses caused by cache coherence
Reduce the miss rate

- **Fast Hit Time + Low Conflict => Victim Cache**
  - Add buffer to place data discarded from cache
- **Hardware Prefetching of Instructions & Data**
  - Prefetching relies on having extra memory bandwidth that can be used without penalty
- **Software Prefetching Data**
- **Compiler Optimizations**
  - Instructions
    - Reorder procedures in memory so as to reduce conflict misses
    - Profiling to look at conflicts (using tools they developed)
- **Data**
  - Merging Arrays, Loop Interchange, Loop Fusion, Blocking
- **Blocking**
Reducing Miss Penalty

• **Read Priority over Write on Miss**
  – Write-through w/ write buffers => RAW conflicts with main memory reads on cache misses
  – Write-back want buffer to hold displaced blocks

• **Early Restart and Critical Word First**
  – Don’t wait for full block to be loaded before restarting CPU
  – Generally useful only in large blocks,
  – Spatial locality => tend to want next sequential word, so not clear if benefit by early restart

• **Non-blocking Caches to reduce stalls on misses**
• **Add a second-level cache**
Reduce the time to hit in the cache

- Small and Simple Caches
- Avoiding Address Translation
  - If index is physical part of address, can start tag access in parallel with translation so that can compare to physical tag
  - Limits cache to page size
- pipelining Cache
Reference

- David A. Patterson, John L. Hennessy, “Computer Organizations & Design: The Hardware/Software Interface”, Morgan Kaufmann
- IDEC, “”, 99