CS 61C: Great Ideas in Computer Architecture
Pipelining & Hazards

Instructors:
John Wawrzynek & Vladimir Stojanovic
http://inst.eecs.Berkeley.edu/~cs61c/fa15
Review: Single-Cycle Processor

• Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     • Formulate Logic Equations
     • Design Circuits
Review: A Single-Cycle Datapath
Review: Controller Implementation

“AND” logic
- opcode
- func
- add
- sub
- ori
- lw
- sw
- beq
- jump

“OR” logic
- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemWrite
- nPCsel
- Jump
- ExtOp
- ALUctr[0]
- ALUctr[1]
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock period is?

<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>

- Clock rate (cycles/second = Hz) = 1/Period (seconds/cycle)
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events

- Clock period is?

<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>

- What can we do to improve clock rate?
- Will this improve performance as well?
  Want increased clock rate to mean faster programs
Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads
• Pipelined laundry takes 3.5 hours for 4 loads!
Pipelining Lessons (1/2)

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3x (8/3.5) v. 4x (8/2) in this example
Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
Execution Steps in MIPS Datapath

1) **IF**F**tch**: Instruction Fetch, Increment PC
2) **Dcd**: Instruction Decode, Read Registers
3) **Exec**: 
   Mem-ref: Calculate Address 
   Arith-log: Perform Operation
4) **Mem**: 
   Load: Read Data from Memory 
   Store: Write Data to Memory
5) **WB**: Write Data Back to Register
Single Cycle Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

- PC
- +4
- Instruction memory
- Registers
- ALU
- Data memory
- Imm
- Rd
- Rs
- Rt
### Pipeline registers

- **Need registers between stages**
  - To hold information produced in previous cycle
More Detailed Pipeline
IF for Load, Store, ...
ID for Load, Store, ...
EX for Load

[Diagram of a computer's execution phase, showing the flow of data through the instruction memory, ALU, and memory units.]
WB for Load – Oops!

Wrong register number!
Corrected Datapath for Load
Pipelined Execution Representation

- Every instruction must take same number of steps, so some stages will idle
  - e.g. MEM stage for any arithmetic instruction
Graphical Pipeline Diagrams

- Use datapath figure below to represent pipeline:

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back
Graphical Pipeline Representation

- RegFile: left half is write, right half is read

Time (clock cycles)

Load
Add
Store
Sub
Or
Pipelining Performance (1/3)

• Use $T_c$ (“time between completion of instructions”) to measure speedup
  
  $T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$

  Equality only achieved if stages are *balanced* (i.e. take the same amount of time)

• If not balanced, speedup is reduced

• Speedup due to increased *throughput*
  
  – *Latency* for each instruction does not decrease
Pipelining Performance (2/3)

• Assume time for stages is
  – 100ps for register read or write
  – 200ps for other stages

<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>

• What is pipelined clock rate?
  – Compare pipelined datapath with single-cycle datapath
Pipelining Performance (3/3)

**Single-cycle**

\[ T_c = 800 \text{ ps} \]
\[ f = 1.25\text{GHz} \]

\[
\begin{align*}
\text{lw} & \ 1, 100($0) \\
\text{lw} & \ 2, 200($0) \\
\text{lw} & \ 3, 300($0)
\end{align*}
\]

**Pipelined**

\[ T_c = 200 \text{ ps} \]
\[ f = 5\text{GHz} \]

\[
\begin{align*}
\text{lw} & \ 1, 100($0) \\
\text{lw} & \ 2, 200($0) \\
\text{lw} & \ 3, 300($0)
\end{align*}
\]
Logic in some stages takes 200ps and in some 100ps. Clk-Q delay is 30ps and setup-time is 20ps. What is the maximum clock frequency at which a pipelined design can operate?

• A: 10GHz
• B: 5GHz
• C: 6.7GHz
• D: 4.35GHz
• E: 4GHz
Adminstrivvia

• HW2 due 10/16 @ 23:59:59

• Project 3-1 due date now 10/21 (release 10/14)
• Project 3-2 due date now 10/28 (release 10/18)
Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy
     (e.g. needed in multiple stages)

2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data read/write

3) *Control hazard*
   - Flow of execution depends on previous instruction
Structural Hazard #1: Single Memory

Trying to read same memory twice in same clock cycle
Solving Structural Hazard #1 with Caches
Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

• Two different solutions have been used:
  1) Split RegFile access in two: Write during 1\textsuperscript{st} half and Read during 2\textsuperscript{nd} half of each clock cycle
     • Possible because RegFile access is VERY fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports

• **Conclusion:** Read and Write to registers during same clock cycle is okay

  *Structural hazards can always be removed by adding hardware resources*
• Consider the following sequence of instructions:

  add $t0, $t1, $t2
  sub $t4, $t0, $t3
  and $t5, $t0, $t6
  or  $t7, $t0, $t8
  xor $t9, $t0, $t10
2. Data Hazards (2/2)

- Data-flow *backwards* in time are hazards

```
add $t0,$t1,$t2
sub $t4,$t0,$t3
and $t5,$t0,$t6
or $t7,$t0,$t8
xor $t9,$t0,$t10
```
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
  - OK that it’s not stored in RegFile yet

add $t0,$t1,$t2

sub $t4,$t0,$t3

and $t5,$t0,$t6

or $t7,$t0,$t8

xor $t9,$t0,$t10
Datapath for Forwarding (1/2)

• What changes need to be made here?
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*
Datapath and Control

- The control signals are pipelined, too
Data Hazard: Loads (1/3)

- **Recall**: Dataflow backwards in time are hazards

\[
lw \ $t0,0($t1) \\
sub \ $t3,$t0,$t2
\]

- Can’t solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)
Data Hazard: Loads (2/3)

- Stalled instruction converted to “bubble”, acts like nop

lw $t0, 0($t1)
sub $t3,$t0,$t2
sub $t3,$t0,$t2
and $t5,$t0,$t4
or $t7,$t0,$t

First two pipe stages stall by repeating stage one cycle later
• Slot after a load is called a *load delay slot*
  – If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  – Letting the hardware stall the instruction in the delay slot is equivalent to putting an explicit `nop` in the slot (except the latter uses more code space)

• **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!
How many cycles (pipeline fill+process+drain) does it take to execute the following code?

lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)

A. 7  B. 9  C. 11  D. 13  E. 14
Code Scheduling to Avoid Stalls

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

```
# Method 1:
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

```
# Method 2:
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t5, 16($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Stall! 13 cycles
Stall! 11 cycles
Break

• (Maybe)
In The News: SanDisk announces ½ PetaByte flash drive

- 512TB of flash memory in 3U of rack space
  - That’s $2^{49}$ bytes
- 780,000 I/O/second
- 7 GB/s sustained bandwidth
3. Control Hazards

• Branch determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on ID stage of branch
• BEQ, BNE in MIPS pipeline
• Simple solution Option 1: Stall on every branch until branch condition resolved
  – Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)
Stall => 2 Bubbles/Clocks

Where do we do the compare for the branch?
Control Hazard: Branching

• Optimization #1:
  – Insert special branch comparator in Stage 2
  – As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  – Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  – Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What’s an efficient way to implement the equality comparison?
One Clock Cycle Stall

Branch comparator moved to Decode stage.
Control Hazards: Branching

• Option 2: *Predict* outcome of a branch, fix up if guess wrong
  – Must cancel all instructions in pipeline that depended on guess that was wrong
  – This is called “flushing” the pipeline

• Simplest hardware if we predict that all branches are NOT taken
  – Why?
Control Hazards: Branching

• Option #3: Redefine branches
  – Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  – New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)

• *Delayed Branch* means *we always execute inst after branch*

• This optimization is used with MIPS
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

or $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:

Delayed Branch

add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or $8, $9, $10
xor $10, $1, $11

Exit:
Control Hazards: Branching

• Notes on Branch-Delay Slot
  – Worst-Case Scenario: put a nop in the branch-delay slot
  – Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    • Re-ordering instructions is common way to speed up programs
    • Compiler usually finds such an instruction 50% of time
    • Jumps also have a delay slot ...
Greater Instruction-Level Parallelism (ILP)

• Deeper pipeline (5 => 10 => 15 stages)
  – Less work per stage ⇒ shorter clock cycle
• Multiple issue “superscalar”
  – Replicate pipeline stages ⇒ 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
• “Out-of-Order” execution
  – Reorder instructions dynamically in hardware to reduce impact of hazards
• *Take CS152 next to learn about these techniques!*
In Conclusion

• Pipelining increases throughput by overlapping execution of multiple instructions in different pipe stages.
• Pipe stages should be balanced for the highest clock rate.
• Three types of pipeline hazard limit performance:
  – Structural (always fixable with more hardware).
  – Data (use interlocks or bypassing to resolve).
  – Control (reduce impact with branch prediction or branch delay slots).