Recall

- Pipelining is parallelizing execution
  - Key to speedups in processors

- Split instruction execution into stages
  - The Five Stages for MIPS execution
  - Add memory to store state
Pipeline Hazards

• Hazard:
  – Situation when next instruction cannot execute in the following clock cycle

• Types of Hazards
  – Structural hazards
  – Control hazards
  – Data hazards
Structural Hazards

- Use the same resource in different ways at the same time and the hardware cannot support the combination
  - Example: Use a single memory for instruction & data
    - If we had more than 4 instructions,
      - 1st instruction will be accessing data
      - 4th instruction fetching the instruction
      - Both need to access the memory in the same clock cycle
  - Since MIPS was designed with two distinct memories, we don’t encounter this problem

=> No hazards
Structural Hazards

- MIPS can easily avoid other structural hazards
  - We can always resolve hazards by waiting
    - pipeline control must detect the hazard
    - take action (or delay action) to resolve hazards
Structural Hazards

- Single Memory is a Structural Hazard

Two memory accesses: If the same memory is used
Control Hazards

• Attempt to make a decision, based on the result of one instruction, before condition is evaluated (Caused in the branch instruction)

• First solution: Pipeline stall (Bubble)
  – Pause (Wait) before continuing the pipeline, until the decision is clear
  • calculate the branch address, update PC during the second stage
Control Hazards

- First solution: Pipeline stall (Bubble)
  - Next instruction halts until condition result is known
  - Like a no-operation is inserted in the 3rd step
  - Next instruction will be executed in 4th step
Control Hazards

• Disadvantages of first solution (Stall)
  – Stall slows down the pipeline

• Second solution: Predict
  – Guess one direction, then backup if wrong
    • Always predict that branch will fail
      – If you are right, pipeline proceeds at full speed (1 clock cycle)
      – If you are wrong, Do pipeline stall (2 clock cycles)
Reduce Delay of Branches

- Move branch execution earlier in pipeline
- Test beq condition using XOR instead of subtraction
  - Faster since no carry is required
Control Hazards

- Second solution: Predict (fig. 6.50)
  - For `beq`, the branch decision is done at cycle 4.
  - The 3 following instructions will be fetched & begin execution
  - If branch is not performed, no time is lost (no stall)
  - If the branch should be performed, these instructions have to be flushed
    - Flushing usually replaces the instruction with `nop` instruction
Control Hazards

- Second solution: Predict
  - Pipeline when branch is not taken
  - No time is wasted
Control Hazards

- Second solution: Predict
  - When branch is taken
  - 2 ns wasted
- Moved test branch decision in 2nd stage
Control Hazards

• Disadvantage of second solution (Predict)
  – Rigid and does not account for the specific branches

• Third solution: Dynamic Branch Prediction
  – Guess depending on previous behavior of branch
    • If right, pipeline proceeds at full speed
    • If wrong, do pipeline stall and change prediction for next time
  – Prediction changes over the lifetime of the program
  – Prediction hardware has ~90% accuracy
    • Cost of mis-prediction is higher
Control Hazards

- Fourth solution: Delayed branch
  - Operation
    - Always execute next instruction immediately after branch instruction, that dependent on the branch, or
    - Try to execute branch first and delay next instruction
  - This is not visible to the programmer
  - Compilers fill ~50% of delays with useful instructions
Data Hazards

• Problem:
  – Instruction depends on the result of previous instruction still in the pipeline
  – Attempt to use an item before it is ready

• Solution: Forwarding (Bypassing):
  – Supply the needed intermediate results to the next instruction’s stages as soon as they are evaluated
  – Get the item early from the internal resources
    • Forwarding:
      – Result is passed forward from an earlier to a later instruction
    • Bypassing:
      – Passing the result by the register file to the desired unit
Data Hazards - Dependencies

Backward dependencies are data hazards
Forward dependencies are not hazards

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)`

<table>
<thead>
<tr>
<th>Value of register $2:</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>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>
<td>–20</td>
</tr>
</tbody>
</table>
Data Hazards- Dependencies

• Example:
  – Problem with starting next instruction before first is finished
  – sub instruction writes into $S2$
  – All following instructions read $S2$
  – Proper value is unavailable until the register is written (in cycle 5)
  – Dependencies that “go backward in time” are data hazards
Data Hazards

- **Example:**

  \[
  \begin{align*}
  &\text{add} \ $s0, \ $t0, \ $t1 \\
  &\text{sub} \ $t2, \ $s0, \ $t3
  \end{align*}
  \]

  - The subtract instruction immediately uses $s0$ that is filled by the add instruction
  - The add instruction doesn’t write the result until the 5th stage
  - Without intervention, a data hazard could severely stall the pipeline

- **Solution:**

  - As soon as the ALU creates the sum for the add, forward it as an input for the subtract
Data Hazards

- **Forwarding**
  - Only valid if the destination stage is later in time than the source stage
  - Output of ALU (EX) of `add` instruction is forwarded to the input of ALU stage for `sub` instruction

```plaintext
Program execution order (in instructions)

add $s0, $t0, $t1

sub $t2, $s0, $t3
```

Diagram:

```
  IF   ID   EX   MEM   WB
  |     |     |     |     |
  2    4    6    8   10

IF: Instruction Fetch
ID: Instruction Decode
EX: Execution
MEM: Memory Access
WB: Write Back
```
Forwarding

- For some instruction types, we need to stall even with forwarding
  - When an R-format instruction comes immediately after a load instruction
  - This is done to prevent backward dependencies

```
lw $s0, 20($t1)
sub $t2, $s0, $t3
```
Forwarding

- Without bubble: Backward (Data hazard)

  Program execution order (in instructions)
  
  * lw $s0, 20($t1)*

  sub $t2, $s0, $t3

  Time

  2  4  6  8  10  12  14

- With bubble: Forward (No hazard)

  Program execution order (in instructions)
  
  * lw $s0, 20($t1)*

  sub $t2, $s0, $t3

  Time

  2  4  6  8  10  12  14
Forwarding

• Solution:
  – Supply inputs to ALU by forwarding results as soon as they are evaluated
  – Don’t wait for the result to be written into register file
    • Register file forwarding
      – Handles read/write to same register
    • ALU forwarding
Forwarding

- Example:
  - $s2$ will have 10 at the beginning and -20 at the end of cycle

```
Example:
- $s2$ will have 10 at the beginning and -20 at the end of cycle

Forwarding
Program execution order (in instruction)

```

```
Program execution order (in instruction)
```

```
Value of register $s2$: CC 1
  CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9
  10   | 10   | 10   | 10 - 20 | -20 | -20 | -20 | -20

Value of EX/MEM: 10
Value of MEM/WB: X

Program execution order (in instruction)
```

```
Program execution order (in instruction)
```

```
sub $s2, $1, $3
and $s2, $2, $5
or $s2, $6, $2
add $s2, $2, $2
sw $s2, 100($2)

Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```

```
Pipeline registers used to forward data
```
Improving Performance

• Exercise:
  – For the following code that resembles a swap procedure:

    ```
    
    # $t1 = Addr v[k]
    lw $t0, 0($t1)   # $t0(temp)= v[k]
    lw $t2, 4($t1)   # $t2 = v[k+1]
    sw $t2, 0($t1)   # v[k] = $t2
    sw $t0, 4($t1)   #v[k+1]= $t0

    ```

  – Draw the pipeline
  – Find the hazards in this code
  – Find out how can to reorder these instructions to avoid stalls
Improving Performance

• What about this order?

  # $t1 = Addr v[k]
  lw $t0, 0($t1)  # $t0(temp)= v[k]
  lw $t2, 4($t1)  # $t2 = v[k+1]
  sw $t0, 4($t1)  #v[k+1]= $t0
  sw $t2, 0($t1)  # v[k] = $t2

• On a machine with forwarding, the reordered sequence will take 4 clock cycles
Recent Trends in Performance

• Super-pipelining
• Super-scalar
• Dynamic pipelining
Super-Pipelining

• Remember:
  – Speedup is related to # stages

• Idea:
  – Make longer pipelines (more stages)
  – Rebalance remaining steps so they are the same length
  – Example: laundry
    – Divide washing into: wash, rinse, & spin
      => 6 stages instead of 4

• Recent microprocessors have >= 8 stages
Super-Scalar

• Idea:
  – Replicate internal components to launch multiple instructions at the same time

• Effect:
  – Instruction execution rate exceeds clock rate (CPI < 1)

• Example: laundry
  – 3 washers
  – 3 dryers
  – 3 assistants to fold
  – 3 assistants to put away laundry

• Example:
  – Vote count
Super-Scalar

• Today’s super-scalar computers have 2-6 instructions in every pipeline stage

• Problem:
  – Difficult to implement if the instruction stream is dependent or doesn’t meet the criteria
Super-Scalar MIPS

• Assumptions
  – 2 instructions issued per clock cycle
    • ALU/Branch instruction, in parallel with
    • Load/Store instruction
  – Need to fetch & decode 64 bits of instructions
  – We examine the instructions & possibly swap them before sending them to the ALU or memory unit to reduce hazards
  – Need extra HW
Super-Scalar MIPS

• Need extra hardware:
  – Separate ALU for address calculation
Super-Scalar MIPS Example

• Example(page 513)

  Loop:  
  \[ \text{lw} \quad \text{$t0, 0($s1)} \]
  \[ \text{addu} \quad \text{$t0, $t0,$s2} \]
  \[ \text{sw} \quad \text{$t0, 0($s1)} \]
  \[ \text{addi} \quad \text{$s1, $s1,-4} \]
  \[ \text{bne} \quad \text{$s1, $zero, Loop} \]

• Assumption:
  - $s1$ contains +16 => Loop iterates 4 times
  - 5 instructions (each needs 4 cycles)
  - Original number of cycles needed = 5 * 4 = 20 cycles

• Exercise:
  - Draw the pipeline for the 4 iterations
Super-Scalar MIPS Example

- When adding another ALU
  => CPI should be ~ .5
- 4 cycles per loop iteration
- Number of cycles = 4 * 4 = 16
  => CPI (Performance) = 16/20 = 0.8
  => CPI value far from optimal

<table>
<thead>
<tr>
<th>ALU/ Branch</th>
<th>Data Transfer</th>
<th>Instruction cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>addi $s1, $s1, -4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>sw, $t0, 4($s1)</td>
<td></td>
</tr>
</tbody>
</table>
Super-Scalar MIPS

• Loop Unrolling Example
  – Multiple copies of the body of the loop are made
  – Different iterations are scheduled together

• Code in Super-scalar MIPS with loop unrolling:
  – (4 copies of loop body)
  – 12 out of 14 instructions work in super-scalar mode
  – Total number of cycles = 8
  – CPI (Performance) = 8/20=0.4

=> Better performance
## Loop Unrolling Example

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addi $s1, $s1, -16</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw $t0, 0($s1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>lw $t2, 8($s1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addu $t1, $t1, $s2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>lw $t3, 4($s1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addu $t2, $t2, $s2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>sw, $t0, 0($s1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addu $t3, $t3, $s2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>sw, $t1, 12($s1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw, $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sw, $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>
Dynamic Pipelining

• Later instructions are executed while waiting for stall to be resolved
• Pipeline divided into 3 major units
  – Instruction fetch & issue unit
    • Send instructions in order
  – Execution units
    • Can execute in parallel (or out-of-order)
  – Commit Unit
    • Send instructions out in order again
Dynamic Pipelining

- Instruction fetch and decode unit
- Reservation station
- Reservation station
- Reservation station
- Reservation station
- Integer
- Integer
- Floating point
- Load/Store
- Functional units
- Execution Unit
- In-order issue
- Out-of-order execute
- In-order commit
Dynamic Pipelining

- Instruction fetch & issue unit:
  - Fetches instructions
  - Decodes instructions
  - Send instruction to the corresponding functional unit

- Execution unit:
  - 5-10 functional unit to hold operands & operators
  - Each functional unit has a unit buffer (reservation station)
  - When all operands are in the buffer & functional unit is ready, result is calculated

- Commit Unit:
  - Decides when it is safe to put result back into register file or memory
Dynamic Pipelining

• The hardware performs the “scheduling”
  – HW tries to find instructions to execute
  – Out of order or parallel execution is possible

• Speculative execution:
  – Combining dynamic scheduling & branch prediction
Dynamic Pipelining

- All modern processors are very complicated
  - DEC Alpha 21264:
    - 9 stage pipeline, 6 instruction issue
  - PowerPC and Pentium:
    - Branch history table
  - Compiler technology is important as well as HW
Pentium Pro & PowerPC 604 pipeline organization
Summary

• Pipelining is a fundamental concept
  – multiple steps using distinct resources

• Utilize capabilities of the Datapath by pipelined instruction processing
  – start next instruction while working on the current one
  – limited by length of longest stage (plus fill/sink)
  – detect and resolve hazards
G4 Processor
Athlon Processor