Chapter 6
Enhancing Performance With Pipelining
Topics

- Explaining pipelining through example
- CPU pipelining
- MIPS instructions & pipelining
- Pipeline hazards
- Recent trends in performance
Introduction

Still in CPU’s Datapath
What is pipelining?

- Implementation technique in which multiple instructions are overlapped in execution
- Real-life pipelining examples?
  - Laundry
  - Factory production lines
  - Car wash
  - Traffic
Pipelining Example: Laundry

- You have 4 loads of cloths to wash:
- Steps (stages) required:
  - Wash
  - Dry
  - Fold
  - Store clothes into drawers
    - Maybe let your roommate help here
- Each stage needs 30 minutes
- We can’t start the next step until the previous step is finished
Pipelining Example: Laundry

- There are 2 approaches to do this job:
  - Sequential (non-pipelined):
    - Wait until the first load is put in drawers to start the next load
  - Pipelined (ASAP):
    - As soon as the washer is empty, start putting the next load, while the first load is put into dryer
Pipelining Example: Laundry

- Sequential Laundry
  - Needs 8 hours for 4 loads
Pipelining Example: Laundry

- Pipelined Laundry:
  - Start work ASAP
  - Needs only 3.5 hours for 4 loads!
Pipelining Example: Laundry

- Pipelined Laundry Observations:
  - At some point, all stages of washing will be operating concurrently
  - Pipelining doesn’t reduce number of stages
    - doesn’t help latency of single task
    - helps throughput of entire workload
  - As long as we have separate resources, we can pipeline the tasks
  - Multiple tasks operating simultaneously use different resources
Pipelining Example: Laundry

- Pipelined Laundry Observations:
  - Speedup due to pipelining depends on the number of stages in the pipeline
  - Pipeline rate limited by slowest pipeline stage
    - If dryer needs 45 minutes, time for all stages has to be 45 minutes to accommodate it
    - Unbalanced lengths of pipe stages reduces speedup
  - Time to “fill” pipeline (Startup) and time to “drain” it (Wind-down) reduces speedup
  - If one load depends on another, we will have to wait (Delay/Stall for Dependencies)
Pipelining Example: Laundry

Problems:

- If we have washer-dryer combination instead of separate washer & dryer, we can’t make the two steps in parallel.
- If your roommate is busy doing something else, you will have to add a stage of going to the drawer and putting your stuff yourself.
  - One resource for folding
  - Another resource for put away

=> Structural Hazard
Pipelining Example: Laundry

Problems:
- If you are cleaning the uniform of a football team, you need to determine whether the detergent & water temperature settings are ok or should be changed, depending on how dirty are the uniforms
  - Stall? Slow
    - Wait until you get the dry uniforms to see if you need to change setup.
    - Repeat until you have the right formula
  - Continue with the next load & check formula later?

=> Control hazards
Pipelining Example: Laundry

Problems:

- In the folding process, if you discover that the load of socks you have need the mate of each of the socks that is still in the washer, you will have to wait until they dry in order to be able to fold

=> Data hazards
CPU Pipelining

- Can we pipeline instruction execution?
- For the following instructions, which resources do you need for each of these steps?
  - store/ load word
  - add/ subtract/ and/ or/ slt
  - branch if equal
CPU Pipelining

- Review: 5 stages of a MIPS instruction
  1. **Fetch** instruction from instruction memory
  2. **Read registers** while decoding instruction
  3. **Execute** operation or calculate address, depending on the instruction type
  4. **Access an operand** from data memory
  5. **Write result** into a register

- We can reduce the cycles to fit the stages.
CPU Pipelining

Example: Resources for Load Instruction
CPU Pipelining

Example: Resources for Load Instruction

1. Fetch instruction from instruction memory (IF)
   - Instruction memory (IM)
2. Read registers while decoding instruction (ID)
   - Register file & decoder (Reg)
3. Execute operation or calculate address, depending on the instruction type (EX)
   - ALU
4. Access an operand from data memory (MEM)
   - Data memory (DM)
5. Write result back into a register (WB)
   - Register file (Reg)
CPU Pipelining

- Note that accessing source & destination registers is performed in two different parts of the cycle.
- We need to decide upon which part of the cycle should reading and writing to the register file take place.
CPU Pipelining: Example

Assumptions:

Only consider the following instructions:

\[ lw, sw, add, sub, and, or, slt, beq \]

Operation times for instruction classes are:

- Memory access: 200 ps
- ALU operation: 200 ps
- Register file read or write: 100 ps

Use a single-cycle (not multi-cycle) model

Clock cycle must accommodate the slowest instruction (200 ps)

Both pipelined & non-pipelined approaches use the same HW components
CPU Pipelining: Example

- Single-Cycle, non-pipelined execution
- Total time for 3 instructions: 24 ns
CPU Pipelining: Example

- Single-cycle, pipelined execution
- Improve performance by increasing instruction throughput
- Total time for 3 instructions = 14 ns
- Each instruction adds 2 ns to total execution time
- Stage time limited by slowest resource (2 ns)
- Assumptions:
  - Write to register occurs in 1st half of clock
  - Read from register occurs in 2nd half of clock
CPU Pipelining Example:

Instructions total times:

- Assumption:
  - Single cycle instruction
  - For non-pipelined
    - Clock must be long enough to allow for the slowest instruction (800 ps)
    - Time between first and fourth instruction = 3 x 800 = 2400 ps
  - For pipelined
    - Each stage should accommodate the longest time for a stage (200 ps)
    - Time between first and fourth instruction = 3 x 200 = 600 ps

### Instruction Fetch Example

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Fetch</th>
<th>Register Read</th>
<th>ALU Operation</th>
<th>Data Access</th>
<th>Register Write</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td>100 ps</td>
<td>800 ps</td>
</tr>
<tr>
<td>sw</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td>200 ps</td>
<td></td>
<td>700 ps</td>
</tr>
<tr>
<td>add, sub, and, or, slt</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td>100 ps</td>
<td>600 ps</td>
</tr>
<tr>
<td>beq</td>
<td>200 ps</td>
<td>100 ps</td>
<td>200 ps</td>
<td></td>
<td></td>
<td>500 ps</td>
</tr>
</tbody>
</table>
CPU Pipelining Example:

- **Assumption:**
  - No delay in the following processor units:
    - MUX
    - Control unit
    - PC access
    - Sign-extend units

- **Slowest resources:**
  - Instruction memory
  - ALU
  - Data memory

- **Theoretically:**
  - Speedup should be equal to number of stages

- **Practically:**
  - Stages are imperfectly balanced
  - Pipelining needs overhead
  - Speedup less than number of stages
CPU Pipelining Example:

- Speedup can be calculated in a formula:
  \[
  \text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Number of pipe stages}}
  \]

- If we have 3 consecutive instructions:
  - Non-pipelined needs \(800 \times 3 = 2400\) ps
  - Pipelined needs 7 stages \(\times 200 = 1400\) ps
  \[=> \text{Speedup} = \frac{2400}{1400} = 1.7\]

- If we have 1003 consecutive instructions:
  - Add more time for 1000 instruction (i.e. 1003 instruction) on the previous example:
    - Non-pipelined total time = \(1000 \times 800 + 2400 = 800,002,400\) ps
    - Pipelined total time = \(1000 \times 200 + 1400 = 200,001,400\) ps
  \[=> \text{Speedup} \sim 3.98 \text{~ (800 ps / 200 ps)}\]
  \[\sim \text{near perfect speedup}\]
  \[=> \text{Performance increases for larger number of instructions (throughput)}\]
Pipelining MIPS Instruction Set

- MIPS was designed with pipelining in mind

=> Pipelining is easy in MIPS:

1. All instructions are the same length
2. Limited instruction format
3. Memory operands appear only in lw & sw instructions
4. Operands must be aligned in memory
Pipelining MIPS Instruction Set

1. All MIPS instruction are the same length
   - Fetch instruction in 1st pipeline stage
   - Decode instructions in 2nd stage
   - If instruction length varies (e.g. IA-32), pipelining will be more challenging
2. MIPS has limited instruction format

- Source register in the same place for each instruction (symmetric)
- 2nd stage can begin reading at the same time as decoding
- If instruction format wasn’t symmetric, stage 2 should be split into 2 distinct stages

=> Total stages = 6 (instead of 5)
3. Memory operands appear only in lw & sw instructions

- We can use the execute stage to calculate memory address
- Access memory in the next stage
- If we needed to operate on operands in memory (e.g. IA-32), stages 3 & 4 would expand to
  - Address calculation
  - Memory access
  - Execute
Pipelining MIPS Instruction Set

4. Operands must be aligned in memory
   - Transfer of more than one data operand can be done in a single stage with no conflicts
   - Need not worry about single data transfer instruction requiring 2 data memory accesses
   - Requested data can be transferred between the CPU & memory in a single pipeline stage
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

• Attempt to use the same resource in two different ways at the same time and the HW cannot support the combination
  
  • Example: If we use a single memory for both 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

- Single Memory is a Structural Hazard

Two memory accesses: If the same memory is used
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

- Forwarding is valid if the destination stage is later in time than the source stage.

Program execution order (in instructions):

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)

Time (in clock cycles):

- CC 1: 10
- CC 2: 10
- CC 3: 10
- CC 4: 10
- CC 5: 10/20
- CC 6: 20
- CC 7: 20
- CC 8: 20
- CC 9: 20

Value of register $2:

1. CC 1: 10
2. CC 2: 10
3. CC 3: 10
4. CC 4: 10
5. CC 5: 20
6. CC 6: 20
7. CC 7: 20
8. CC 8: 20
9. CC 9: 20

Backward dependencies are data hazards.
Forward dependencies are not hazards.
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:

\[ \text{add } \$s0, \$t0, \$t1 \]
\[ \text{sub } \$t2, \$s0, \$t3 \]

- 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, replacing the value from $s0$.
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
Forwarding

- Without bubble: Backward (Data hazard)

- With bubble: Forward (No hazard)
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
Forwarding

Example:
- Why not take the values from the registers?
Control (Branch) Hazards

- Attempt to make a decision, based on the result of one instruction, before condition is evaluated (Caused in the branch instruction)
- First solutions: Pipeline stall (Bubble) on branch
  - Pause (wait) before continuing the pipeline, until the decision is clear
    - Calculate the branch address, update PC during the second stage
  - Disadvantage:
    - The cost of this option is too high for most computers
Control (Branch) Hazards

- Example: Here we have backward referencing (Hazard)
Control (Branch) Hazards

- Pipeline stall (Bubble) on branch
- Errata!! and instead of add
Control (Branch) Hazards

- 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)
  - Disadvantage of Predict
    - Rigid. Does not account for the individuality of specific branch instructions
Control (Branch) Hazards

- Pipeline when branch is not taken
  - If branch is not performed, no time is lost (no stall)
Control (Branch) Hazards

- Pipeline when branch is taken
  - The instruction should be flushed
  - Flushing usually replaces the instruction with *nop* instruction
Control (Branch) Hazards

Third solution: Dynamic HW Prediction

- Make your guess depending on previous behavior of each branch
  - If you are right, pipeline proceeds at full speed
  - If you are wrong, Do pipeline stall and change your prediction for next time
- Prediction may change over the lifetime of the program
- Prediction HW has ~ 90% accuracy
- Cost of mis-prediction is higher
Control (Branch) Hazards

- Fourth solution: Delayed branch
  - Used by MIPS
    - Always execute the next instruction immediately after the delayed branch instruction, that isn’t affected by the branch, or
    - Try to execute the branch first and delay the instruction
    - This is not visible to the programmer
    - Compilers typically fill about 50% of branch delays with useful instructions

```
beq $1, $2, 40
add $4, $5, $6
lw $3, 300($0)
```

Program execution order (in instructions)
Pipelined Datapath

**Basic Idea:**
- Resources for Different Stages should be Independent
- We need to add some registers & other HW to be able to split the datapath into stages
Pipelined Datapath

- **Example:** *add* Instruction
  - **Terminology:**
    - Shading indicates that the element is used by the instruction
    - Shading on the *right* means reading
    - Shading on the *left* means writing
    - For register file
      - Hashed means not used at the corresponding half of the cycle
  - **Stages & resources required:**
    - IF (Instruction fetch) - Instruction memory
    - ID (Instruction decode / register file read) - Register file
    - EX (Execution) - ALU
    - MEM (Memory access) - Not needed in this instruction
    - WB (Write back) - Register file

```
ad $s0, $t0, $t1
```
Pipelined Datapath

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?

lw $10, 20($1)
sub $11, $2, $3
Pipelined Datapath

Another way to represent pipelining

Program Flow

Time

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB
Pipelined Datapath

- Example: three lw instructions
Pipelined Datapath

- Pipelined version of the datapath
  - Pipeline registers separate each pipeline stage
  - Registers must be wide enough to store all data corresponding to the lines going through them
  - Each register is named after the stages it separates
    - IF/ID: 64 bits (32 instruction, 32 next PC value)
    - ID/EX: 128 bits (32 sign-extended value, 32 data1, 32 data2, 32 next PC value)
    - EX/MEM: 97 bits (32 data2, 32 modified PC value, 32 ALU result, 1 Zero signal)
    - MEM/WB: 64 bits (32 ALU result, 32 Read data)
Pipelined Datapath

- lw instruction execution
  - Step 1: Instruction fetch
Pipelined Datapath

- lw instruction execution
  - Step 2: Instruction decode & read register file
Pipelined Datapath

- lw instruction execution
  - Step 3: Execute (calculate address of word to be loaded)
Pipelined Datapath

- *lw* instruction execution
  - Step 4: Memory Access (get value stored in the address)
Pipelined Datapath

- lw instruction execution
  - Step 5: Write Back
Pipelined Datapath

- lw instruction execution
  - Step 1: Instruction fetch
Pipelined Datapath

- lw instruction execution
  - Step 2: Instruction decode & read register file
Pipelined Datapath

- sw instruction execution
  - Step 3: Execute (calculate address of word to be stored)
Pipelined Datapath

- `sw` instruction execution
  - Step 4: Memory Access (store value in the data memory)
Pipelined Datapath

- sw instruction execution
  - Step 5: Write Back (Idle/ no action performed)
Pipelined Datapath

- Bug in the design
  - Destination register # will be that of another following instruction in the pipeline
Pipelined Datapath

- Correcting the design Bug
  - Pass the register number from ID/EX, EX/MEM, MEM/WB to be used in the WB stage
Pipelined Datapath

- Portion of datapath used by all 5 stages
  - No conflict
Pipelined Datapath

- Multiple-Cycle datapath
Pipelined Datapath

- Traditional multiple-Cycle datapath
Pipelined Datapath

- A single-clock-cycle datapath corresponding to clock cycle 5 of the pipeline
Pipelined Datapath

- Pipeline datapath including control signals (no time to discuss control signals)
Improving Performance

Exercise:

For the following code that resemble the 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 we 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 Improving 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 $\geq$ 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 HW:
  - Separate ALU for address calculation
Super-Scalar MIPS Example

Example (page 513)

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

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

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

**Code in Super-Scalar MIPS**

- 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><em>Loop:</em></td>
<td><em>lw $t0, 0($s1)</em></td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1, -4</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, <em>Loop</em></td>
<td><em>sw, $t0, 4($s1)</em></td>
<td>4</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

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

In-order issue

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Execution Unit

Functional units

Integer

Integer

Floating point

Load/Store

Out-of-order execute

In-order commit

Commit unit

Load/Store

Integer
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
Intel Pentium 4 Micro-architecture
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
Homework!

- Problems, pp. 455-463
  - In class:
    - Problem 6.39
  - Homework
    - Problems: 6.2, 6.3, 6.4, 6.6, 6.14, 6.17