Pipelining, Superscalar, Multiprocessors

CPS 104

Admin
- Homework due Wednesday April 25
- Final May 1, 2pm-5pm
- Review (either in class Wednesday – overview OR Costi will have one)

Pipelining: Its Natural!
- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

Sequential Laundry
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP
- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
Overview of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Cycle time: time it takes to execute the longest step
- Keep all the steps so they have similar length
- This is the essence of the multiple cycle processor

The advantages of the multiple cycle processor:
- Cycle time is much shorter
- Different instructions take different number of cycles to complete
  - Load takes five cycles
  - Jump only takes three cycles
- Allows a functional unit to be used more than once per instruction

Multiple Cycle Processor

- MCP: If a functional unit is used more than once per instruction -> cannot pipeline -> lower performance

The Five Stages of Load

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- WrB: Write the data back to the register file

Key Ideas Behind Instruction Execution Pipelining

- Overlap execution of instructions
- The load instruction has 5 stages: Ifetch, Reg-Fetch / I-Decode, Execute, Memory-Access, Register Write-Back.
  - Five independent functional units to work on each stage
    - Each functional unit is used only once
  - The 2nd load can start as soon as the 1st finishes its Ifetch stage
  - Each load still takes five cycles to complete. latency is still 5 cycles
  - The throughput is much higher; CPI is 1 with ~1/5 cycle time.
  - Instructions start before the previous ones are completed

Pipelining the Load Instruction

- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the WrB stage
- One instruction enters the pipeline every cycle
- One instruction comes out of the pipeline (completed) every cycle
- The "Effective" Cycles per Instruction (CPI) is 1; ~1/5 cycle time

The Four Stages of R-type

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Register access and Instruction Decode
- Exec: ALU operates on the two register operands
- WrB: Write the ALU output back to the register file
**Pipelining the R-type and Load Instruction**

- Cycle 1: Fetch
- Cycle 2: Decode
- Cycle 3: Execute
- Cycle 4: Write

We have a problem called a pipeline conflict or resource hazard:
- Two instructions try to write to the register file at the same time!

**Clock Cycle**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
</tr>
<tr>
<td>Load</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WrLoad</td>
<td>Mem</td>
<td>WrLoad</td>
<td>Mem</td>
<td>WrLoad</td>
</tr>
<tr>
<td>R-type</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
</tr>
<tr>
<td>R-type</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
<td>Mem</td>
<td>WrR-type</td>
</tr>
</tbody>
</table>

- **Important Observation**
  - Each functional unit can only be used once per instruction
  - Each functional unit must be used in the same stage for all instructions:
    - Load uses Register File’s Write Port during its 4th stage
    - R-type uses Register File’s Write Port during its 4th stage

**How to solve this pipeline hazard?**

Delay R-type’s write by one cycle:
- Now R-type instructions also use Reg File’s write port at Stage 5
- Mem stage is a NO-OP stage: nothing is being done. Effective CPI?

**The Four Stages of Load**

- Ifetch: Instruction Fetch
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

**A Pipelined Datapath**

- Ifetch: Instruction Fetch
- Reg/Dec: Registers Fetch and Instruction Decode
- Execute: ALU compares the two register operands
- Mem: If the registers we compared in the Exec stage are the same, write the branch target address into the PC
The Instruction Fetch Stage
- Location 10: lw $1, 100($2) $1 <- Mem[($2) + 0x100]

The Decode / Register Fetch Stage
- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

Load's Address Calculation Stage
- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

Load's Memory Access Stage
- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]

Load's Write Back Stage
- Location 10: lw $1, 0x100($2) $1 <- Mem[($2) + 0x100]
A More Extensive Pipelining Example

Clock Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
<td>Fetch</td>
</tr>
<tr>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
<td>Reg/Dec</td>
</tr>
<tr>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
<td>Exec</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
<td>WrB</td>
</tr>
</tbody>
</table>

End of Cycle 4: Load's Mem, R-type's Exec, Store's Reg, Beq's Ifetch
End of Cycle 5: Load's WrB, R-type's Mem, Store's Exec, Beq's Reg
End of Cycle 6: R-type's WrB, Store's Mem, Beq's Exec
End of Cycle 7: Store's WrB, Beq's Mem

Data Hazards
- So far we ignored instructions dependencies, but in a real machine one must deal with dependencies.
- Example:
  - sub $2, $1, $3
  - and $12, $2, $5 # $12 depends on the result in $2
  - or $13, $6, $2 # but $2 is updated 3 clock
  - add $14, $2, $2 # cycles later.
  - sw $15, 100($2) # We have a problem!!

Data Hazard Solution: Register Forwarding

Single Cycle, Multiple Cycle, vs. Pipeline

Pipelining Summary
- Most modern processors use pipelining
- Pentium 4 has 24 (35) stage pipeline!
- Alpha 21164 has 7 stages
- Pipelining creates more headaches for exceptions, etc...
- Pipelining augmented with superscalar capabilities

Superscalar Processors
- Key idea: execute more than one instruction per cycle
- Pipelining exploits parallelism in the “stages” of instruction execution
- Superscalar exploits parallelism of independent instructions
- Example Code:
  - sub $2, $1, $3
  - and $12, $2, $5
  - or $13, $6, $2
  - add $3, $3, $2
  - sw $15, 100($2)
- Superscalar Execution
  - add $2, $1, $3 and $12, $2, $5
  - add $3, $3, $2
  - sw $15, 100($2)
Superscalar Processors

- **Key Challenge**: Finding the independent instructions
- Instruction level parallelism (ILP)
- **Option 1**: Compiler
  - Static scheduling (Alpha 21064, 21164; UltraSPARC I, II; Pentium)
- **Option 2**: Hardware
  - Dynamic Scheduling (Alpha 21264; PowerPC; Pentium Pro, 3, 4)
  - Out-of-order instruction processing

Instruction Level Parallelism

- **Problems:**
  - Program structure: branch every 4-8 instructions
  - Limited number of registers
  - Static scheduling: compiler must find and move instructions from other basic blocks
  - Dynamic scheduling: Hardware creates a big "window" to look for independent instructions
    - Must know branch directions before branch is executed!
    - Determines true dependencies.
- **Example Code:**
  ```
  sub $2, $1, $3
  and $12, $3, $2
  or $2, $6, $4
  add $3, $3, $2
  sw $15, 100($2)
  ```

Exposing Instruction Level Parallelism

- **Branch prediction**
  - Hardware can remember if branch was taken
  - Next time it sees the branch it uses this to predict outcome
- **Register renaming**
  - Indirection! The CS solution to almost everything
  - During decode, map register name to real register location
  - New location allocated when new value is written to reg.
- **Example Code:**
  ```
  sub $2, $1, $3 # writes $2 = $p1
  and $12, $3, $2 # reads $p1
  or $2, $6, $4 # writes $2 = $p3
  add $3, $3, $2 # reads $p3
  sw $15, 100($2) # reads $p3
  ```

CPU design Summary

- **Disadvantages of the Single Cycle Processor**
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- **Multiple Clock Cycle Processor**
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- **Pipeline Processor**
  - Natural enhancement of the multiple clock cycle processor
  - Each functional unit can only be used once per instruction
  - If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions
- **Pipeline Control**
  - Each stage’s control signal depends ONLY on the instruction that is currently in that stage

Additional Notes

- All Modern CPUs use pipelines.
- Many CPUs have 8-12 pipeline stages.
- The latest generation processors (Pentium-4, PowerPC G4, SUN’s UltraSPARC) use multiple pipelines to get higher speed (Superscalar design).
- Now, Parallel Architectures...
Parallel Computation: Why and Why Not?

**Pros**
- Performance
- Cost-effectiveness (commodity parts)
- Smooth upgrade path
- Fault Tolerance

**Cons**
- Difficult to parallelize applications
- Requires automatic parallelization or parallel program development
- Software! AAAAAHHH!

Simple Problem

for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]

Split the loops
- Independent iterations
for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
for i = 1 to N
    sum = sum + A[i]

Message Passing Architectures

- IBM SP-2, Intel Paragon
- Cluster of workstations
- Server Racks

Small Scale Shared Memory Multiprocessors

- Small number of processors connected to one shared memory
- Memory is equidistant from all processors (UMA)
- Kernel can run on any processor (symmetric MP)
- Intel dual/quad Pentium
- Multicore

Cache Coherence Problem (Initial State)

Cache Coherence Problem (Step 1)
**Cache Coherence Problem (Step 2)**

- P1
- P2
- Add r1, r2, r4
- St x, r1
- Time
- BUS
- Main Memory
- Id r2, x

**Cache Coherence Problem (Step 3)**

- P1
- P2
- Add r1, r2, r4
- St x, r1
- Time
- BUS
- Main Memory
- Id r2, x

**Large Scale Shared Memory Multiprocessors Shared**

- 100s to 1000s of nodes (processors) with single shared physical address space
- Use General Purpose Interconnection Network
- Still have cache coherence protocol
- Use messages instead of bus transactions
- No hardware broadcast
- Network Interface
- Cray T3D, T3E, Compaq EV7, SUN ES3000

**Directory Example**

- P1
- Directory Node
- P2
- P3
- Time
- M: Request
- S: Invalidate
- M: Response
- Ad: Ack