Pipelines

Today's topics:
- Evidence suggests there is some rust on this topic
  - hence spend a week and move on
- also need some common terminology
- Attempt to present the ideal issues
  - with some discussion on why ideal isn't reality.

---

Pipelining

- Computational assembly line
  - each step does a small fraction 1/pipeline_depth of the job
  - concurrent execution of pipeline_depth instructions
    - performance is all about parallelism
- Vertical vs. Horizontal concurrency
- Pipeline stage – 1 step in an N step pipe
  - 1 cycle per stage
    - synchronous design – slowest stage sets clock rate
    - laminar is the target
- Simple model

![](image)

---

Pipeline Benefit = Performance

- Ideal performance
  - time-per-instruction = unpiped_instruction_time/#stages
    - asymptotic – overheads count
      - +10% typically achieved
  - 2 ways to view this performance enhancement
    - logical
      - work on several instructions at once
      - parallelism
        - average IPC reduced
    - physical
      - shorter stages = increased frequency

---

Other Pipeline Benefits

- HW mechanism
  - hidden from the SW so invisible to the user
  - just viewed as a benefit
- No programming impact
  - unless user needs the ultimate in performance
  - usually left up to compiler scheduling & optimization
- Pipelines everywhere
  - key keep on Moore's law curve in the 80's
  - 90's just moved to multiple pipelines
- Frequency wars
  - push pipeline depth to lunatic fringe
    - problems
      - power x frequency
      - overheads make ideal performance a bit optimistic
Consider MIPS64

- 5 steps in instruction execution
  - fetch, decode, execute, mem, write-back
- Remember the ISA

## 5 steps in instruction execution

1. Fetch: Loads and stores, 8-immed
2. Decode: Alu op, Cond branches (cz) and imm16, Jump reg, Jcc, reg reg
5. Write-back: Offset added to PC

### Example 5-stage Data-path

- **IF**
  - PC path needs to change bit in the pipeline version.
- **ID**
  - Pre-IF
  - Next PC
- **EX**
  - Ir: opcode, RS1, RS2, RD, imm16, function
  - Wbmux value
- **MEM**
  - PC+4
  - Immediate data: 16 or 25 bits
- **WB**
  - ALU out, SMD, mux selector indices, R vs. W command

### InterStage Registers

- **Pre-IF**
  - Next PC
- **IF:ID**
  - PC+4
  - Ir: opcode, RS1, RS2, RD, imm16, function
  - Wbmux value
- **EX:MEM**
  - PC+4
  - Immediate data: 16 or 25 bits
- **MEM:WB**
  - ALU out, SMD, mux selector indices, R vs. W command

Stages vary by Instructions

- **Stage 3**
  - Reg-reg or calculate effective address or branch target
    - for any instruction
    - only one role
- **Stage 4**
  - only active on Load/Store/Jump/Branch
    - LMD = Mem[ALUoutput]
    - Mem[ALUoutput] = SMD
    - next PC = ALUoutput if condition
    - JUMP - no condition
- **Stage 5**
  - Reg-Reg
    - Regs[IRi+2]=ALUoutput
  - Reg-Immediate
    - Regs[IRi+2]=ALUoutput
  - Load
    - Regs[IRi+2] = memory data return
How real was that?

- Depends
  - real for simple architectures
    - woefully over simplified for higher performance architectures
  - not optimized
    - 2 ALU's
      - IF and EX - but ALU's are cheap so who cares?
    - Harvard architecture
      - separate instruction and data memories
        - typical at L1 - but unified below that
    - Sx frequency for five stages
      - slowed down by inter-stage register overhead
  - Data-path is only part of the architecture
    - largest bit in terms of area
    - easiest bit in terms of getting it right
    - control path
      - FSM or microcode or both?

Control vs. Data Example

- Look at a few typical components

<table>
<thead>
<tr>
<th>Component</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Addr, Data, Data1</td>
<td>R32, R31, R30, Data</td>
</tr>
<tr>
<td>Counter</td>
<td>Data, Data, Carry, Carry1</td>
<td>X, Inc, dec, br, R30, R31</td>
</tr>
<tr>
<td>Scratch</td>
<td>Data, Data</td>
<td>Data, Data</td>
</tr>
<tr>
<td>ALU</td>
<td>Data, Data, Data, Carry, Carry, OP</td>
<td></td>
</tr>
<tr>
<td>RTU</td>
<td>Write-Data, Data, Select</td>
<td></td>
</tr>
<tr>
<td>Decoder</td>
<td>Data, Data, Data, Select</td>
<td></td>
</tr>
</tbody>
</table>

Sequential/Combinational | clock oriented

Control Path

- Each component has control points
  - registers read or output enable
  - mux/demux select lines
  - memory R vs. W
  - XU - opcode
- What vs. When
  - when controlled by a clock
    - SDR vs. BSR
  - what controlled by FSM or uCode control point values
- Note
  - book ignores this for the most part
    - fine in a way
      - tends to consume a small amount of area and power
    - BUT tends to be the major problem
      - in terms of getting it right!

Example: FSM for a simple Add

- Rx <- Rx + Ry

Decoder will determine R31 & R32
- R31 & R32
  - will drive appropriate output enable
  - selected line
- Decoder will drive "X" code to the ALU
  - use ALU from the decoded IR instruction
- Note: steps 4 and 5 could
  - have been combined since
  - the ALU may be
  - dependent on PRT information
Full Control Scenario

After step 2, the instruction class is known
each class may require different control point assertions
Note that only the load requires all 5 cycles - dummy step 4 goes as all instructions finish at the same time.

Pipeline Parallelism

- Best case - execute 5 instructions at once
  - Note pipeline fill and flush overhead
  - In stead state
    - 5x frequency → Ideal speedup

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Problem
  - consider single I & D memory
    - step 4 & 5 have a resource conflict

Pipeline Characteristics

- Latency
  - time it takes for an instruction to complete
  - worse w/ pipeline since latch delay added to critical path
    - dominant feature if lots of exceptions
    - steady state doesn’t last for long
    - branch miss predicts, cache misses, real exceptions

- Throughput
  - dominant feature if steady state is common
    - compiler tries hard to make this true
    - e.g. no
      - cache misses
      - register misses
      - speculation failures
      - real exceptions

Example

- Unpipelined
  - 5 steps: 50, 50, 60, 50, 50 ns respectively
  - total 260 ns
- Turn it into a pipelined design
  - 10 ns of “laminarity” penalty
  - 5 ns delay due to latches
    - set-up, hold, and fall through delays
- Hence
  - must run at slowest stage rate/clock = 65 ns
  - speedup 260/64 = 4x
    - rather than idealized 5x
Pipeline Hair

- **Laminarity is hard**
  - depends a lot on FO4 budget
    - 20+ FO4 is somewhat easy
    - 13- has proven to be problematic

- **Extra resources**
  - each stage needs its own
    - list all possible instruction resource needs
    - separate by stage
    - each stage needs its private set

- **Example**
  - PC modification can’t use same ALU as arithmetic ops
  - IF & Mem can’t access same memory

Pipeline Memory Issues

- **More instructions on the fly**
  - increased memory pressure & bandwidth requirements
    - Nf for N stage pipeline

- **Key issue w/ memory**
  - it’s slow
    - bigger memories are slower and consume more power
      - tiled improves latency but not power

- **Fixes**
  - Harvard architecture
    - independent roles
    - access patterns are different
      - optimization opportunity
  - multi-level cache & memory hierarchy
  - speculative prefetch
  - pipeline the memory system
    - works for both cache and main

Hazards & Dependencies

- **Consider a pair of instructions**
  - R5 = R2 + R3; R3 = R5 + R6
    - write back of R5 happens in stage 5
    - R5 value needed by stage 3
    - OOPS

- **Enter bypass and stalls**
  - value actually known at end of stage 3
  - used on next cycle in stage 3
  - send/bypass value to stage 4 and to beginning of stage 3
    - more logic and more control
      - add mux delay \(\rightarrow\) catch-22
      - dependencies must be checked
      - time each register goes KA-CHING
    - impact on both data and control paths

3 Types of Hazards

- **Structural**
  - resource contention of different pipeline stages
    - register read in ID or register write in WB
    - 2 ported register file
      - typical with up to 2 reads and one write \(\rightarrow\) 3 ported RR and IW
        - superscalar makes this worse

- **Data**
  - dependency for either register source or destination

- **Control**
  - PC incremented or computed
    - branch and jump effect
  - exceptions \(\rightarrow\) go somewhere else
    - e.g. exception handler
    - not as bad with an in-order execution style
      - total pain with out-of-order execution
        - more on this later
Example Pipeline Activity

- From pipeline stage perspective

<table>
<thead>
<tr>
<th>Stage</th>
<th>PC → PC + 4</th>
<th>R1 → Mem(PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>PC ← PC</td>
<td>R1 ← R2</td>
</tr>
<tr>
<td>EX</td>
<td>F</td>
<td>R1 ← S2</td>
</tr>
<tr>
<td>MEM</td>
<td>PC ← PC</td>
<td>ALU → ALU</td>
</tr>
<tr>
<td>WB</td>
<td>R1 ← ALUOut</td>
<td>R1 ← WMAR</td>
</tr>
</tbody>
</table>

Pipeline Activity

- From Instruction class perspective

<table>
<thead>
<tr>
<th>Stage</th>
<th>ALU instruction</th>
<th>Load or Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>R ← MEM(PC)</td>
<td>R ← MEM(PC)</td>
</tr>
<tr>
<td>ID</td>
<td>A ← B(R1); B ← R2; PC1 ← PC; PC ← R1; R2 ← R1</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>A ← B(R1); B ← R2; PC1 ← PC; PC ← R1; R2 ← R1</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>ALUout ← ALUout; MEM ← MEM(MAR) ← SMAR</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>R1 ← ALUout</td>
<td>R1 ← WMAR or M3(MAR) ← SMAR</td>
</tr>
</tbody>
</table>

Hazards & Stalls

- Extra resources mitigate
  - data → bypass logic
  - structural → duplicate resources
  - control → predict and speculate

- When it fails
  - stall

  * Ideal pipeline speedup compromised

- More realistic scenario
  - not all stages are necessary for every instruction
  - implementation increases control path complexity

Pipeline: Resource View

- Structural Hazard?
**Calculating Stall Effects**

Pipeline Speedup = \( \frac{\text{Average instruction time without pipelining}}{\text{Average instruction time with pipelining}} \)

Pipeline Speedup = \( \frac{\text{unstpped cycle time}}{\text{piped cycle time}} \)

\( \text{Ideal CPI} = \frac{\text{unstpped CPI}}{\text{piped CPI}} \)

Therefore

\( \text{Pipeline Speedup} = \frac{\text{unstpped cycle time}}{\text{piped cycle time}} \times \frac{\text{unstpped CPI}}{\text{piped CPI}} \)

**Conclusion: Beware of Overhead**

- Cycle time
  - reduces w/ increased # of stages
  - but latch insertion adds to latency
  - size of inter-stage registers is large
  - increased power due to lots of bits moved and stored

- Stall effects
  - the deeper the pipeline
  - reduced probability that nothing went wrong
  - e.g. reduction from 1/N speedup ideal

- High frequency
  - active power linear w/ frequency
  - stall restart is a problem w/ very high frequencies
  - e.g. Prescott and Northwood

- Amdahl's Law
  - not everything benefits
  - no guessing – you have to run the sim's

**Calculating Further**

However

\( \text{piped CPI} = \text{Ideal CPI} \times \text{Pipeline stall cycles} \times (1 - \text{average stalls per instruction}) \)

Then if perfect balance: no overhead and cycle times equal

\[ \text{Speedup} = \frac{\text{CPI Unpiped}}{1 + \text{Pipeline stall cycles per instruction}} \]

If laminar then unpiped CPI = pipeline depth, hence

\[ \text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stalls per instruction}} \]

Similar derivations for clock cycles are also possible