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 $1 / \text{pipeline_depth}$ 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 set clock rate
    - laminar is the target
- Simple model

\[ \text{Clock} \]

\[ \text{Stage 1 Latch} \rightarrow \text{CL} \rightarrow \text{Stage 2 Latch} \rightarrow \text{CL} \rightarrow \text{Stage n Latch} \rightarrow \text{CL} \]
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
        • albeit in different stages of their execution
      » 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 are 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 α frequency
        - overheads make ideal performance a bit optimistic
Consider MIPS64

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs1</th>
<th>rd</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 6 10 11 15 16 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I-type: loads and stores, R-immed
ALU ops, Cond branch (rs1 and rd unused), Jump reg, J & Link reg

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs1</th>
<th>rs2</th>
<th>rd</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 6 10 11 15 16 20 21 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-type: reg-reg ALU ops (func=op)
read/write special regs, and moves

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
</table>

J-type: Jump and Link, Jump, TRAP, and RFE

Stages vary by Instructions

- Stage 3
  - Xeq 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 w/ condition
    - JUMP – no condition

- Stage 5
  - Reg-Reg
    - Regs[IR_{16..20}] ← ALUoutput
  - Reg – Immediate
    - Regs[IR_{14..15}] ← ALUoutput
  - Load
    - Regs[IR_{11..15}] ← memory data return
Example 5-stage Data-path

**Pre-IF**
- Next PC

**IF:ID**
- PC+4
- IR: opcode, RS1, RS2, RD, imm16, function
- Wbmux value

**ID:EX**
- PC+4
- IR1: Amux_sel, Bmux_sel, ALUop, Wbmux_sel, R/Wmem, Mmux_sel
- immediate data: 16 or 26 bits

**EX:Mem**
- ALUout, SMD, mux selector indices, R vs. W command

**MiWB**
- ALUout, LMD
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
      » 5x 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>Address, Data, Data</td>
<td>RAS, CAS, R/W, Out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add, Radd</td>
</tr>
<tr>
<td>Counter</td>
<td>Data, Data, Carry, Carry</td>
<td>Clear, up, down, Ld, Out</td>
</tr>
<tr>
<td>Tri-state buffers</td>
<td>Data, Data</td>
<td>Out</td>
</tr>
<tr>
<td>Register</td>
<td>Data, Data,</td>
<td>Ld, Out</td>
</tr>
<tr>
<td>ALU</td>
<td>LData, RData, Data, Carry</td>
<td>OP</td>
</tr>
<tr>
<td>Max</td>
<td>Many-Data, Data</td>
<td>Select</td>
</tr>
<tr>
<td>DeMux</td>
<td>Many-Data, Data</td>
<td>Select</td>
</tr>
<tr>
<td>1 cycle Barrel Shifter</td>
<td>Data, Data</td>
<td>Shift Amount</td>
</tr>
</tbody>
</table>

Sequential/Combinational clock oriented
Control Path

• Each component has control points
  • register: load or output enable
  • mux/demux: select lines
  • memory: R vs. W
  • XU – optcode

• What vs. When
  • when controlled by a clock
    » SDR vs. DDR
  • 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

No matter what the instruction is, we have to do this

Really a selection - but here we know it’s an R type so we take this path

Decoder will demux RS1 & RS2 fields to drive appropriate output enables of selected source registers. Don’t really know the instruction type so we load them anyway plus the immediate value as well.

Note: steps 4 and 5 could have been combined since for an ADD there are no dependencies
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 pads so 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 $\Rightarrow$ ideal speedup

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock 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>i+1</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>i+2</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>i+3</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>i+4</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>
</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 F04 budget
    » 20+ F04 is somewhat easy
    » 13- has proven to be problematic

• **Extra resources**
  - each stage needs it’s own
    » design drill
      • list all possible instruction resource needs
      • separate by stage
      • each stage needs it’s 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
    » Nx 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 → catch-22
  » dependencies must be checked
    • time cash 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 arith op is 2 reads and one write → 3 ports: 2R and IW
        – superscalar makes this worse

• Data
  • dependency for either register source or destination

• Control
  • PC incremented or computed
    » branch and jump effect
  • exceptions → go somewhere else
    » e.g. exception handler
    » not so 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 Unit</th>
<th>Memory</th>
<th>Data Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>PC ← PC + 4</td>
<td>IR ← Mem[PC]</td>
<td>A ← Rs1; B ← Rs2</td>
</tr>
<tr>
<td>ID</td>
<td>PC1 ← PC</td>
<td>IR1 ← IR</td>
<td>DMAR ← A + (IR1)H ## IR16.31 or ALUout ← A op B or ALUout ← A op (IR1)H ## IR16.31 or ALUout ← PC1 + (IR1)H ## IR16.31 or cond ← (Rs1 op 0); SMDR ← B</td>
</tr>
<tr>
<td>EX</td>
<td></td>
<td></td>
<td>ALUout ← ALUout</td>
</tr>
<tr>
<td>MEM</td>
<td>if (cond) then PC ← ALUout</td>
<td>LMDR ← MEM[DMAR] or MEM[DMAR] ← SMDR</td>
<td>ALUout ← ALUout</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td>Rd ← ALUout or Rd ← LMDR</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>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IR ← MEM[PC]; PC ← PC+4</td>
<td>IR ← MEM[PC]; PC ← PC+4</td>
<td>IR ← MEM[PC]; PC ← PC+4</td>
</tr>
<tr>
<td>ID</td>
<td>A ← Rs1; B ← Rs2; PC1 ← PC; IR1 ← IR</td>
<td>A ← Rs1; B ← Rs2; PC1 ← PC; IR1 ← IR</td>
<td>A ← Rs1; B ← Rs2; PC1 ← PC; IR1 ← IR</td>
</tr>
<tr>
<td>EX</td>
<td>ALUout ← A op B or ALUout ← A op (IR1)H ## (IR16.31)</td>
<td>DMAR ← A + (IR1)H ## IR16.31 or SMDR ← B (if it’s a store)</td>
<td>ALUout ← PC1 + (IR1)H ## IR16.31 or cond ← Rs1 op 0</td>
</tr>
<tr>
<td>MEM</td>
<td>ALUout ← ALUout</td>
<td>LMDR ← MEM[DMAR] or MEM[DMAR] ← SMDR</td>
<td>if (cond) then PC ← ALUout</td>
</tr>
<tr>
<td>WB</td>
<td>Rd ← ALUout</td>
<td>Rd ← LMDR (if it’s a load)</td>
<td></td>
</tr>
</tbody>
</table>

- note potential stage holes where nothing much happens
- note pre-decode (stages 1 & 2) - same for all
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?
### Stall Creates Pipeline Bubble

1d=inst0, Mem → Reg → Mem → Reg

inst1 (not 1d), Mem → Reg → Mem

inst2, Mem → Reg → Mem → Reg

inst3, Mem → Reg → Mem

---

### Calculating Stall Effects

**Pipeline Speedup**

\[
\text{Pipeline Speedup} = \frac{\text{Average instruction time without pipelining}}{\text{Average instruction time with pipelining}}
\]

\[
\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{unpiped CPI}}{\text{piped CPI}}
\]

**Ideal CPI**

\[
\text{Ideal CPI} = \frac{\text{unpiped CPI}}{\text{Pipeline Depth}}
\]

**Therefore**

\[
\text{Pipeline Speedup} = \frac{\text{unpiped cycle time}}{\text{piped cycle time}} \times \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{piped CPI}}
\]
Calculating Further

However

\[ \text{piped CPI} = \text{Ideal CPI} \times \text{Pipeline stall cycles} = 1 + \text{average stalls per instruction} \]

Then if perfect balance: no overhead and cycle times equal

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

If lamina 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

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