CIS 371
Computer Organization and Design

Unit 5: Pipelining

Based on slides by Prof. Amir Roth & Prof. Milo Martin

Readings

- P&H
  - Chapter 4 (4.5 – 4.8)

This Unit: Pipelining

- Performance
  - Processor performance equation
- Multicycle datapath
- Basic Pipelining
- Data Hazards
  - Software interlocks and scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control Hazards
  - Branch prediction
- Pipelined multi-cycle operations

Pre-Class Exercises

- You have a washer, dryer, and “folder”
  - Each takes 30 minutes per load
  - How long for 1 load in total?
  - How long for two loads of laundry?
  - How long for 100 loads of laundry?

- Now assume:
  - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
  - How long for one load in total?
  - How long for two loads of laundry?
  - How long for 100 loads of laundry?
Pre-Class Exercises Answers

- You have a washer, dryer, and “folder”
  - Each takes 30 minutes per load
  - How long for one load in total? 90 minutes
  - How long for two loads of laundry? 90 + 30 = 120 minutes
  - How long for 100 loads of laundry? 90 + 30*99 = 3060 min

- Now assume:
  - Washing takes 30 minutes, drying 60 minutes, and folding 15 min
  - How long for one load in total? 105 minutes
  - How long for two loads of laundry? 105 + 60 = 165 minutes
  - How long for 100 loads of laundry? 105 + 60*99 = 6045 min

Processor Performance Equation

- Multiple aspects to performance: helps to isolate them
- Program runtime = latency = “seconds per program” = \( \text{Instructions/program} \times \text{cycles/instruction} \times \text{seconds/cycle} \)
- **Instructions per program**: “dynamic instruction count”
  - Runtime count of instructions executed by the program
  - Determined by program, compiler, instruction set architecture (ISA)
- **Cycles per instruction**: “CPI” (typical range: 2 to 0.5)
  - On average, how many cycles does an instruction take to execute?
  - Determined by program, compiler, ISA, micro-architecture
- **Seconds per cycle**: clock period, length of each cycle
  - Inverse metric: cycles per second (Hertz) or cycles per ns (Ghz)
  - Determined by micro-architecture, technology parameters
- For low latency (better performance) minimize all three
  - Difficult: often pull against one another

Cycles per Instruction (CPI)

- **CPI**: Cycle/instruction for on average
  - \( \text{IPC} = 1/\text{CPI} \)
    - Used more frequently than CPI
    - Favorited because “bigger is better”, but harder to compute with
  - Different instructions have different cycle costs
    - E.g., “add” typically takes 1 cycle, “divide” takes >10 cycles
    - Depends on relative instruction frequencies
- **CPI example**
  - A program executes equal: integer, floating point (FP), memory ops
  - Cycles per instruction type: integer = 1, memory = 2, FP = 3
  - What is the CPI? \( (33\% \times 1) + (33\% \times 2) + (33\% \times 3) = 2 \)
  - **Caveat**: this sort of calculation ignores many effects
    - Back-of-the-envelope arguments only
Improving Clock Frequency

- **Faster transistors**

- Micro-architectural techniques
  - **Multi-cycle processors**
    - Break each instruction into small bits
    - Less logic delay -> improved clock frequency
    - Different instructions take different number of cycles
    - CPI > 1
  - **Pipelined processors**
    - As above, but overlap parts of instruction (parallelism!)
    - Faster clock, but CPI can still be around 1

**Single-Cycle & Multi-Cycle Datapath**

Recall: The Sequential Model

- Basic structure of all modern ISAs
- Processor logically executes loop at left
- **Program order**: total order on dynamic insns
  - Order and named storage define computation
- Convenient feature: **program counter (PC)**
  - Insn itself at memory[PC]
  - Next PC is PC++ unless insn says otherwise
- **Atomic**: insn X finishes before insn X+1 starts
  - Can break this constraint physically (pipelining)
  - But must maintain illusion to preserve programmer sanity

**Single-Cycle Datapath**

- Single-cycle datapath: true “atomic” fetch/execute loop
  - Fetch, decode, execute one complete instruction every cycle
  - "Hardwired control": opcode decoded to control signals directly
  - Low CPI: 1 by definition
  - Long clock period: to accommodate slowest instruction
Multi-Cycle Datapath

- **Multi-cycle datapath**: attacks slow clock
  - Fetch, decode, execute one complete insn over multiple cycles
  - Micro-coded control: "stages" control signals
  - Allows insns to take different number of cycles (main point)
  - Opposite of single-cycle: short clock period, high CPI (think: CISC)

Single-cycle vs. Multi-cycle

- **Single-cycle datapath**: Fetch, decode, execute one complete instruction every cycle
  - Low CPI: 1 by definition
  - Long clock period: to accommodate slowest instruction

- **Multi-cycle datapath**: attacks slow clock
  - Fetch, decode, execute one complete insn over multiple cycles
  - Allows insns to take different number of cycles
  - Opposite of single-cycle: short clock period, high CPI (think: CISC)

Single-cycle vs. Multi-cycle Performance

- **Single-cycle**
  - Clock period = 50ns, CPI = 1
  - Performance = **50ns/insn**

- **Multi-cycle**
  - Has opposite performance split of single-cycle
  - Shorter clock period
  - Higher CPI

  - **Branch**: 20% (3 cycles), load: 20% (5 cycles), ALU: 60% (4 cycles)
  - Clock period = **11ns**, CPI = (20%*3)+(20%*5)+(60%*4) = 4
    - Why is clock period 11ns and not 10ns?
    - Performance = **44ns/insn**

Pipelining Basics

Aside: CISC makes perfect sense in multi-cycle datapath
Performance: Latency vs. Throughput

- **Latency (execution time):** time to finish a fixed task
- **Throughput (bandwidth):** number of tasks in fixed time
  - Different: exploit parallelism for throughput, not latency (e.g., bread)
  - Often contradictory (latency vs. throughput)
    - Will see many examples of this
    - Choose definition of performance that matches your goals
      - Scientific program? Latency, web server: throughput?

Example: move people 10 miles
- Car: capacity = 5, speed = 60 miles/hour
- Bus: capacity = 60, speed = 20 miles/hour
- Latency: car = 10 min, bus = 30 min
- Throughput: car = 15 PPH (count return trip), bus = 60 PPH

Fastest way to send 1TB of data? (at 100+ mbits/second)

Latency versus Throughput

- Can we have both low CPI and short clock period?
  - Not if datapath executes only one insn at a time
- Latency and throughput: two views of performance ...
  - (1) at the program level and (2) at the instructions level
- Single instruction latency
  - Doesn’t matter: programs comprised of billions of instructions
  - Difficult to reduce anyway
- Goal is to make programs, not individual insns, go faster
  - Instruction throughput → program latency
  - Key: exploit inter-insn parallelism

Pipelining

- Important performance technique
  - Improves instruction throughput rather instruction latency
- Begin with multi-cycle design
  - When insn advances from stage 1 to 2, next insn enters at stage 1
  - Form of parallelism: "insn-stage parallelism"
  - Maintains illusion of sequential fetch/execute loop
  - Individual instruction takes the same number of stages + But instructions enter and leave at a much faster rate
- Laundry analogy

5 Stage Multi-Cycle Datapath
5 Stage Pipeline: Inter-Insn Parallelism

- **Pipelining**: cut datapath into N stages (here 5)
  - One insn in each stage in each cycle
  - Clock period = \( \text{MAX}(T_{\text{insn-mem}}, T_{\text{regfile}}, T_{\text{ALU}}, T_{\text{data-mem}}) \)
  - Base CPI = 1: insn enters and leaves every cycle
  - Actual CPI > 1: pipeline must often stall
  - Individual insn latency increases (pipeline overhead), not the point

Some More Terminology

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: “superscalar” (later in the semester, briefly)

- **In-order pipeline**: insns enter execute stage in order
  - Alternative: “out-of-order” (later in the semester, very briefly)

- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Contemporary high-performance cores have ~15 stage pipelines

**Instruction Convention**

- Different ISAs use inconsistent register orders
- Some ISAs (for example MIPS)
  - Instruction destination (i.e., output) on the left
    - add $1, $2, $3 means $1 \leftarrow $2+$3
  - Some ISAs (for example MIPS)
    - Instruction destination (i.e., output) on the right
      - add $1, $2, $3 means $1 \leftarrow $2+$3
      - ld 8(r5), $4 means mem[r5+8] \rightarrow $4
      - st $4, 8(r5) means $4 \rightarrow \text{mem}[r5+8]
- Will try to specify to avoid confusion, next slides MIPS style
Pipeline Example: Cycle 1

• 3 instructions

Pipeline Example: Cycle 2

Pipeline Example: Cycle 3

Pipeline Example: Cycle 4

• 3 instructions
Pipeline Example: Cycle 5

Pipeline Example: Cycle 6

Pipeline Example: Cycle 7

Pipeline Diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: X means lw $4, 0($5) finishes execute stage and writes into M latch at end of cycle 4
Example Pipeline Perf. Calculation

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn

- Multi-cycle
  - Branch: 20% (3 cycles), load: 20% (5 cycles), ALU: 60% (4 cycles)
  - Clock period = 11ns, CPI = (20%*3)+(20%*5)+(60%*4) = 4
  - Performance = 44ns/insn

- 5-stage pipelined
  - Clock period = 12ns approx. (50ns / 5 stages) + overheads
  - CPI = 1 (each insn takes 5 cycles, but 1 completes each cycle)
    + Performance = 12ns/insn
  - Well actually ... CPI = 1 + some penalty for pipelining (next)
    - CPI = 1.5 (on average insn completes every 1.5 cycles)
    - Performance = 18ns/insn
  - Much higher performance than single-cycle or multi-cycle

Q1: Why Is Pipeline Clock Period ...

- ... > (delay thru datapath) / (number of pipeline stages)?
  - Three reasons:
    - Latches add delay
    - Pipeline stages have different delays, clock period is max delay
    - [Later:] Extra datapaths for pipelining (bypassing paths)
  - These factors have implications for ideal number pipeline stages
    - Diminishing clock frequency gains for longer (deeper) pipelines

Q2: Why Is Pipeline CPI...

- ... > 1?
  - CPI for scalar in-order pipeline is 1 + stall penalties
  - Stalls used to resolve hazards
    - Hazard: condition that jeopardizes sequential illusion
    - Stall: pipeline delay introduced to restore sequential illusion

- Calculating pipeline CPI
  - Frequency of stall * stall cycles
  - Penalties add (stalls generally don't overlap in in-order pipelines)
    - 1 + stall-freq_1*stall-cyc_1 + stall-freq_2*stall-cyc_2 + ...

- Correctness/performance/make common case fast (MCCF)
  - Long penalties OK if they happen rarely, e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

Data Dependences, Pipeline Hazards, and Bypassing
Dependences and Hazards

- **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
    - Not a bad thing, programs would be boring without them
    - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline

- **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order cannot be externally visible
  - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance

Why Does Every Insn Take 5 Cycles?

- Could/should we allow `add` to skip M and go to W? No
  - It wouldn’t help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine `add` follows `lw`

Example Structural Hazard

- **Structural hazards**: resource needed twice in one cycle
  - Example: unified instruction & data memories (caches)
  - Solutions:
    - Separate instruction/data memories (caches)
    - Redesign cache to allow 2 accesses per cycle (slow, expensive)
    - Stall pipeline
Data Hazards

- Let’s forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine...
  - But it wasn’t a real program
  - Real programs have data dependences
    - They pass values via registers and memory

Dependent Operations

- Independent operations
  - add $3, $2, $1
  - add $6, $5, $4
- Would this program execute correctly on a pipeline?
  - add $3, $2, $1
  - add $6, $5, $4
- What about this program?
  - add $3, $2, $1
  - lw $4, 8($5)
  - addi $6, 1, $3
  - sw $3, 8($7)

Data Hazards

- Would this “program” execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - add is writing its result into $3 in current cycle
    - lw read $3 two cycles ago → got wrong value
    - addi read $3 one cycle ago → got wrong value
  - sw is reading $3 this cycle → maybe (depending on regfile design)

Memory Data Hazards

- Are memory data hazards a problem for this pipeline? No
  - lw following sw to same address in next cycle, gets right value
  - Why? Data mem read/write always take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is three stages after register read
  - Can only read a register value three cycles after writing it
Fixing Register Data Hazards

- Can only read register value three cycles after writing it

**Option #1: make sure programs don’t do it**
- Compiler puts two independent insns between write/read insn pair
  - If they aren’t there already
- Independent means: “do not interfere with register in question”
  - Do not write it: otherwise meaning of program changes
  - Do not read it: otherwise create new data hazard
- **Code scheduling:** compiler moves around existing insns to do this
  - If none can be found, must use **nops** (no-operation)

- This is called **software interlocks**
  - **MIPS:** Microprocessor w/out Interlocking Pipeline Stages

Software Interlock Example

```
add $3,$2,$1
nop
lw $4,8($3)
add $6,$2,$8
addi $3,$5,4
```

- Can any of last three insns be scheduled between first two
  - **sw $7,8($3)**? No, creates hazard with **add $3,$2,$1**
  - **add $6,$2,$8**? Okay
  - **addi $3,$5,4**? No, **lw** would read $3 from it
- Still need one more insn, use **nop**
  - **add $3,$2,$1**
  - **add $6,$2,$8**
  - **lw $4,8($3)**
  - **sw $7,8($3)**
  - **addi $3,$5,4**

Software Interlock Performance

- Assume
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- For software interlocks, let’s assume:
  - 20% of insns require insertion of 1 **nop**
  - 5% of insns require insertion of 2 **nops**

- Result:
  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  - 30% more insns (30% slowdown) due to data hazards

Hardware Interlocks

- Problem with software interlocks? Not compatible
  - Where does 3 in “read register 3 cycles after writing” come from?
    - From structure (depth) of pipeline
  - What if next MIPS version uses a 7 stage pipeline?
    - Programs compiled assuming 5 stage pipeline will break
- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
- Two aspects to this
  - Detecting hazards
  - Fixing hazards
Detecting Data Hazards

- Compare input register names of insn in D stage with output register names of older insns in pipeline
  
  \[ \text{Stall} = (D.IR.\text{RegSrc1} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc1} == M.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == M.IR.\text{RegDest}) \]

Fixing Data Hazards

- Prevent D insn from reading (advancing) this cycle
  - Write \textit{nop} into X.IR (effectively, insert \textit{nop} in hardware)
  - Also reset (clear) the datapath control signals
  - Disable D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Hardware Interlock Example: cycle 1

\[ \text{Stall} = (D.IR.\text{RegSrc1} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc1} == M.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == M.IR.\text{RegDest}) = 1 \]

Hardware Interlock Example: cycle 2

\[ \text{Stall} = (D.IR.\text{RegSrc1} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == X.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc1} == M.IR.\text{RegDest}) \lor (D.IR.\text{RegSrc2} == M.IR.\text{RegDest}) = 1 \]
### Hardware Interlock Example: cycle 3

- **Register File**
- **Data Memory**

#### Pipeline Control Terminology
- Hardware interlock maneuver is called **stall** or **bubble**
- Mechanism is called **stall logic**
- Part of more general **pipeline control** mechanism
  - Controls advancement of insns through pipeline
- Distinguish from **pipelined datapath control**
  - Controls datapath at each stage
  - Pipeline control controls advancement of datapath control

#### Pipeline Diagram with Data Hazards
- Data hazard stall indicated with **d**
  - Stall propagates to younger insns

<table>
<thead>
<tr>
<th></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><strong>add $3, $2, $1</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $4, 0($3)</strong></td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $6, 4($7)</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Doing something "smarter" opens up a can of worms:

<table>
<thead>
<tr>
<th></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><strong>add $3, $2, $1</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $4, 0($3)</strong></td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $6, 4($7)</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Hardware Interlock Performance
- As before:
  - Branch: 20%, load: 20%, store: 10%, other: 50%

- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (i.e., insertion of 1 nop)
  - 5% of insns require 2 cycle stall (i.e., insertion of 2 nops)

<table>
<thead>
<tr>
<th></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><strong>add $3, $2, $1</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $4, 0($3)</strong></td>
<td>F</td>
<td>d*</td>
<td>d*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $6, 4($7)</strong></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- CPI = \(1 + 0.20\times1 + 0.05\times2 = 1.3\)
- So, either CPI stays at 1 and #insns increases 30% (software)
- Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
- Same difference

- Anyway, we can do better
**Observation!**

- Technically, this situation is broken
  - `lw $4,8($3)` has already read $3 from regfile
  - `add $3,$2,$1` hasn't yet written $3 to regfile
- But fundamentally, everything is OK
  - `lw $4,8($3)` hasn't actually used $3 yet
  - `add $3,$2,$1` has already computed $3

**Bypassing**

- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**

**WX Bypassing**

- What about this combination?
  - Add another bypass path and MUX (multiplexor) input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

**ALUinB Bypassing**

- Can also bypass to ALU input B
WM Bypassing?

- Does WM bypassing make sense?
  - Not to the address input (why not?)
  - But to the store data input, yes

Pipeline Diagrams with Bypassing

- If bypass exists, “from”/“to” stages execute in same cycle
  - Example: MX bypass

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  
  - Example: WX bypass

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - Example: WM bypass

<table>
<thead>
<tr>
<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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Can you think of a code example that uses the WM bypass?
Have We Prevented All Data Hazards?

- No. Consider a "load" followed by a dependent "add" insn
- Bypassing alone isn't sufficient!
- Hardware solution: detect this situation and inject a stall cycle
- Software solution: ensure compiler doesn't generate such code

Stalling on Load-To-Use Dependences

\[
\text{Stall} = (X.IR.\text{Operation} == \text{LOAD}) \&\&
(\ (D.IR.\text{RegSrc1} == X.IR.\text{RegDest}) \|\ |
\ ((D.IR.\text{RegSrc2} == X.IR.\text{RegDest}) \&\& (D.IR.\text{Op} != \text{STORE}))
\]
Reducing Load-Use Stall Frequency

• Use compiler scheduling to reduce load-use stall frequency
  • As done for software interlocks, but for performance not correctness

<table>
<thead>
<tr>
<th></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>add $3, $2, $1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $4, $3(3)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $6, $4, 1</td>
<td>F</td>
<td>D*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $8, $3, $1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assume
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 50% of loads are followed by dependent instruction
    - require 1 cycle stall (i.e., insertion of 1 \textit{nop})

- Calculate CPI
  - \( \text{CPI} = 1 + (1 \times 20\% \times 50\%) = 1.1 \)

Pipelining and Multi-Cycle Operations

• What if you wanted to add a multi-cycle operation?
  - E.g., 4-cycle multiply
  - \textbf{P}: separate output latch connects to W stage
  - Controlled by pipeline control finite state machine (FSM)

A Pipelined Multiplier

• Multiplier itself is often pipelined, what does this mean?
  - Product/multiplicand register/ALUs/latches replicated
  - Can start different multiply operations in consecutive cycles
What about Stall Logic?

Stall = (OldStallLogic) ||
(D.IR.RegSrc1 == P0.IR.RegDest) || (D.IR.RegSrc2 == P0.IR.RegDest) ||
(D.IR.RegSrc1 == P1.IR.RegDest) || (D.IR.RegSrc2 == P1.IR.RegDest) ||
(D.IR.RegSrc1 == P2.IR.RegDest) || (D.IR.RegSrc2 == P2.IR.RegDest)

Pipeline Diagram with Multiplier

- Allow independent instructions

- Stall subsequent dependent instructions:

Multiplier Write Port Structural Hazard

- What about...
  - Two instructions trying to write register file in same cycle?
  - Structural hazard!
- Must prevent:

```
<table>
<thead>
<tr>
<th></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>mul $4,$3,$5</td>
<td>F</td>
<td>D</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $6,$1,1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $5,$6,$10</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Solution? stall the subsequent instruction

```
<table>
<thead>
<tr>
<th></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>mul $4,$3,$5</td>
<td>F</td>
<td>D</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $6,$1,1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $5,$6,$10</td>
<td>F</td>
<td>D*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Preventing Structural Hazard

- Fix to problem on previous slide:
  Stall = (OldStallLogic) ||
  (D.IR.RegDest "is valid" &&
  D.IR.Operation != MULT &&
  P0.IR.RegDest "is valid")
More Multiplier Nasties

- What about...
  - Mis-ordered writes to the same register
  - Software thinks `add` gets $4$ from `addi`, actually gets it from `mul`

<table>
<thead>
<tr>
<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>mul $4$, $3$, $5$</td>
<td>F</td>
<td>D</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>addi $4$, $1$, $1$</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>add $10$, $4$, $6$</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Common? Not for a 4-cycle multiply with 5-stage pipeline
  - More common with deeper pipelines
  - In any case, must be correct

Corrected Pipeline Diagram

- With the correct stall logic
  - Prevent mis-ordered writes to the same register
  - Why two cycles of delay?

<table>
<thead>
<tr>
<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>mul $4$, $3$, $5$</td>
<td>F</td>
<td>D</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>addi $4$, $1$, $1$</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>add $10$, $4$, $6$</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Multi-cycle operations complicate pipeline logic

Preventing Mis-Ordered Reg. Write

- Fix to problem on previous slide:
  Stall = (OldStallLogic) ||
  ((D.IR.RegDest == X.IR.RegDest) && (X.IR.Operation == MULT))

Pipelined Functional Units

- Almost all multi-cycle functional units are pipelined
  - Each operation takes $N$ cycles
  - But can start initiate a new (independent) operation every cycle
  - Requires internal latching and some hardware replication
  - A cheaper way to add bandwidth than multiple non-pipelined units

<table>
<thead>
<tr>
<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulf $0$, $1$, $2$</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf $3$, $4$, $5$</td>
<td>F</td>
<td>D</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- One exception: int/FP divide: difficult to pipeline and not worth it

<table>
<thead>
<tr>
<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>divf $0$, $1$, $2$</td>
<td>F</td>
<td>D</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divf $3$, $4$, $5$</td>
<td>F</td>
<td>D</td>
<td>s*</td>
<td>s*</td>
<td>s*</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>E/</td>
<td>W</td>
</tr>
</tbody>
</table>

- **s* = structural hazard, two insns need same structure**
- ISAs and pipelines designed to have few of these
- Canonical example: all insns forced to go through M stage
Control Dependences and Branch Prediction

Branch Recovery

- **Branch recovery**: what to do when branch is actually taken
  - Insns that will be written into D and X are wrong
  - **Flush them**, i.e., replace them with *nops*
    + They haven’t had written permanent state yet (regfile, DMem)
      - Two cycle penalty for taken branches (non-fast branch)

What about Branches?

- **Branch speculation**: Could just stall to wait for branch outcome (two-cycle penalty)
- **Fetch past branch insns before branch outcome is known**
  - Default: assume "**not-taken**" (at fetch, can’t tell it’s a branch)

Branch Speculation and Recovery

**Correct**:

- `addi r1,1→r3`
- `bnez r3,targ`
- `st r6→[r7+4]`
- `mul r8,r9→r10`

**Recovery**: (speculative)

- `addi r1,1→r3`
- `bnez r3,targ`
- `st r6→[r7+4]`
- `mul r8,r9→r10`
- `targ:add r4,r5→r4`
## Branch Performance

- Back of the envelope calculation
  - **Branch:** 20%, load: 20%, store: 10%, other: 50%
  - Say, 75% of branches are taken

- CPI = 1 + 20% * 75% * 2 =
  
  \[ 1 + 0.20 \times 0.75 \times 2 = 1.3 \]
  
  - Branches cause 30% slowdown
    - Even worse with deeper pipelines

- Can we do better than assuming branch is not taken?

## Big Idea: Speculative Execution

- Speculation: “risky transactions on chance of profit”

- **Speculative execution**
  - Execute before all parameters known with certainty
  - **Correct speculation**
    + Avoid stall, improve performance
  - **Incorrect speculation (mis-speculation)**
    - Must abort/flush/squash incorrect instructions
    - Must undo incorrect changes (recover pre-speculation state)

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct instructions to execute next?

## Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+4
  - Can actively guess other targets: **dynamic branch prediction**

- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated instructions
    - Hopefully haven’t modified permanent state (Regfile, DMem)
      + Happens naturally in in-order 5-stage pipeline

## Dynamic Branch Prediction

- **Dynamic branch prediction**: hardware guesses outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction**
Dynamic Branch Prediction Components

- Step #1: is it a branch?
  - Easy after decode...
- Step #2: is the branch taken or not taken?
  - Direction predictor (applies to conditional branches only)
    - Predicts taken/not-taken
- Step #3: if the branch is taken, where does it go?
  - Easy after decode...

Branch Direction Prediction

- Learn from past, predict the future
  - Record the past in a hardware structure
- Direction predictor (DIRP)
  - Map conditional-branch PC to taken/not-taken (T/N) decision
    - Individual conditional branches often biased or weakly biased
      - 90%+ one way or the other considered “biased”
      - Why? Loop back edges, checking for uncommon conditions
- Branch history table (BHT): simplest predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - What about aliasing?
    - Two PC with the same lower bits?
      - No problem, just a prediction!

Branch History Table (BHT)

- Branch history table (BHT): simplest direction predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - Problem: inner loop branch below
    - for (i=0;i<100;i++)
      - for (j=0;j<3;j++)
        - // whatever
          - Two “built-in” mis-predictions per inner loop iteration
          - Branch predictor “changes its mind too quickly”

Two-Bit Saturating Counters (2bc)

- Two-bit saturating counters (2bc) [Smith 1981]
  - Replace each single-bit prediction
    - \( (0,1,2,3) = (N,n,t,T) \)
  - Adds “hysteresis”
    - Force predictor to mis-predict twice before “changing its mind”
    - One mispredict each loop execution (rather than two)
      - Fixes this pathology (which is not contrived, by the way)
      - Can we do even better?
Correlated Predictor

**Correlated (two-level) predictor** [Patt 1991]
- Exploits observation that branch outcomes are correlated
- Maintains separate prediction per (PC, BHR) pairs
  - Branch history register (BHR): recent branch outcomes
- Simple working example: assume program has one branch
  - BHT: one 1-bit DIRP entry
  - BHT+2BHR: \(2^2 = 4\) 1-bit DIRP entries
    - Why didn't we do better?
      - BHT not long enough to capture pattern

Recall: Fastest and Slowest Leaf Nodes

**Correlated Predictor Design**

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global is better, captures local patterns for tight loop branches

- Design choice II: how many history bits (BHR size)?
  - Tricky one
    + Given unlimited resources, longer BHRs are better, but...
      - BHT utilization decreases
        - Many history patterns are never seen
        - Many branches are history independent (don’t care)
      - PC xor BHR allows multiple PCs to dynamically share BHT
      - BHR length < \(\log_2\) (BHT size)
    - Predictor takes longer to train
    - Typical length: 8–12

**Try 3 bits of history**

- 2^3 DIRP entries per pattern

**Pattern**

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Prediction</th>
<th>Outcome</th>
<th>Result?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NNN</td>
<td>N</td>
<td>N</td>
<td>Wrong</td>
</tr>
<tr>
<td>2</td>
<td>NNT</td>
<td>T</td>
<td>N</td>
<td>Wrong</td>
</tr>
<tr>
<td>3</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>4</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>5</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>6</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>7</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>8</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>9</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>10</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>11</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>12</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
</tbody>
</table>

+ No mis-predictions after predictor learns all the relevant patterns!

**2^3 DIRP entries per pattern**

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Prediction</th>
<th>Outcome</th>
<th>Result?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NNN</td>
<td>N</td>
<td>N</td>
<td>Wrong</td>
</tr>
<tr>
<td>2</td>
<td>NNT</td>
<td>T</td>
<td>N</td>
<td>Wrong</td>
</tr>
<tr>
<td>3</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>4</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>5</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>6</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>7</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>8</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>9</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>10</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>11</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
<tr>
<td>12</td>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>Wrong</td>
</tr>
</tbody>
</table>

**Expected**:

- Let’s just consider the leaves
  - Same depth, similar instruction count -> similar runtime

**Some of the fastest leaves** (all ~24):

- L = Left, R = Right
  - LL:LLLLLLLLLLLLL
  - LL:LLLLLLLLLLLLL (or any with one “R”)
  - LL:RRRRRLLLRLRLRL
  - LL:RRRRRLLLRLRLRL
  - LL:RRRRRLLLRLRLRL

**Runtime distribution** - Intel Core2

**Some of the slowest leaves**:

- L = Left, R = Right
  - L:RRRRRRRRRRRRRRRRR (~62)
  - L:RRRRRRRRRRRRRRRRR (~56)
  - L:RRRRRRRRRRRRRRRRR (~56)
Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling 1993]
  - Attacks correlated predictor BHT capacity problem
  - Idea: combine two predictors
    - **Simple BHT** predicts history independent branches
    - **Correlated predictor** predicts only branches that need history
  - **Chooser** assigns branches to one predictor or the other
  - Branches start in simple BHT, move mis-prediction threshold
  - Correlated predictor can be made **smaller**, handles fewer branches
  - 90–95% accuracy

When to Perform Branch Prediction?

- **Option #1: During Decode**
  - Look at instruction opcode to determine branch instructions
  - Can calculate next PC from instruction (for PC-relative branches)
    - One cycle “mis-fetch” penalty **even if branch predictor is correct**

- **Option #2: During Fetch?**
  - How do we do that?

Revisiting Branch Prediction Components

- **Step #1**: is it a branch?
  - Easy after decode... during fetch: **predictor**
- **Step #2**: is the branch taken or not taken?
  - **Direction predictor** (as before)
- **Step #3**: if the branch is taken, where does it go?
  - **Branch target predictor** (**BTB**) supplies target PC if branch is taken

Branch Target Buffer (BTB)

- As before: learn from past, predict the future
  - Record the past branch targets in a hardware structure

- **Branch target buffer (BTB):**
  - “guess” the future PC based on past behavior
  - “Last time the branch X was taken, it went to address Y”
    - “So, in the future, if address X is fetched, fetch address Y next”

- **Operation**
  - A small RAM: address = PC, data = target-PC
  - Access at Fetch **in parallel** with instruction memory
    - predicted-target = BTB[hash(PC)]
  - Updated at X whenever target ! = predicted-target
    - BTB[hash(PC)] = target
  - Hash function is just typically just extracting lower bits (as before)
  - Aliasing? No problem, this is only a prediction
Branch Target Buffer (continued)

- At Fetch, how does insn know it’s a branch & should read BTB? It doesn’t have to...
  - ...all insns access BTB in parallel with Imem Fetch
- Key idea: use BTB to predict which insn are branches
  - Implement by “tagging” each entry with its corresponding PC
  - Update BTB on every taken branch insn, record target PC:
    - BTB[PC].tag = PC, BTB[PC].target = target of branch
  - All insns access at Fetch in parallel with Imem
    - Check for tag match, signifies insn at that PC is a branch
      - Predicted PC = (BTB[PC].tag == PC) ? BTB[PC].target : PC+4

Why Does a BTB Work?

- Because most control insns use direct targets
  - Target encoded in insn itself → same “taken” target every time
- What about indirect targets?
  - Target held in a register → can be different each time
  - Two indirect call idioms
    - Dynamically linked functions (DLLs): target always the same
    - Dynamically dispatched (virtual) functions: hard but uncommon
  - Also two indirect unconditional jump idioms
    - Switches: hard but uncommon
    - Function returns: hard and common but...

Return Address Stack (RAS)

- Return address stack (RAS)
  - Call instruction? RAS[TopOfStack++] = PC+4
  - Return instruction? Predicted-target = RAS[--TopOfStack]
  - Q: how can you tell if an insn is a call/return before decoding it?
    - Accessing RAS on every insn BTB-style doesn’t work
    - Answer: another predictor (or put them in BTB marked as “return”)
      - Or, pre-decode bits in insn mem, written when first executed

Putting It All Together

- BTB & branch direction predictor during fetch
  - If branch prediction correct, no taken branch penalty
Branch Prediction Performance

- Dynamic branch prediction
  - 20% of instruction branches
  - Simple predictor: branches predicted with 75% accuracy
    - CPI = 1 + (20% * 25% * 2) = 1.1
  - More advanced predictor: 95% accuracy
    - CPI = 1 + (20% * 5% * 2) = 1.02
- Branch mis-predictions still a big problem though
  - Pipelines are long: typical mis-prediction penalty is 10+ cycles
  - For cores that do more per cycle, predictions most costly (later)

Pipeline Depth

- Trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) “super-pipelining”
  - Core1/2: 14 stages
- Increasing pipeline depth
  - Increases clock frequency (reduces period)
    - But double the stages reduce the clock period by less than 2x
    - Decreases IPC (increases CPI)
      - Branch mis-prediction penalty becomes longer
      - Non-bypassed data hazard stalls become longer
    - At some point, actually causes performance to decrease, but when?
      - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
    - “Optimal” pipeline depth is program and technology specific

Summary

- Performance
- Multicycle datapath
- Basic Pipelining
- Data Hazards
  - Software interlocks and scheduling
  - Hardware interlocks and stalling
  - Bypassing
- Control Hazards
  - Branch prediction
- Pipelined multi-cycle operations