Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Processor Architecture

Lab Schedule

Activities

- **Today**
  - Processor Architecture
- **Wednesday**
  - Network Programming
- **Fri / Mon / Wed**
  - **Lab 12 – Network Programming**

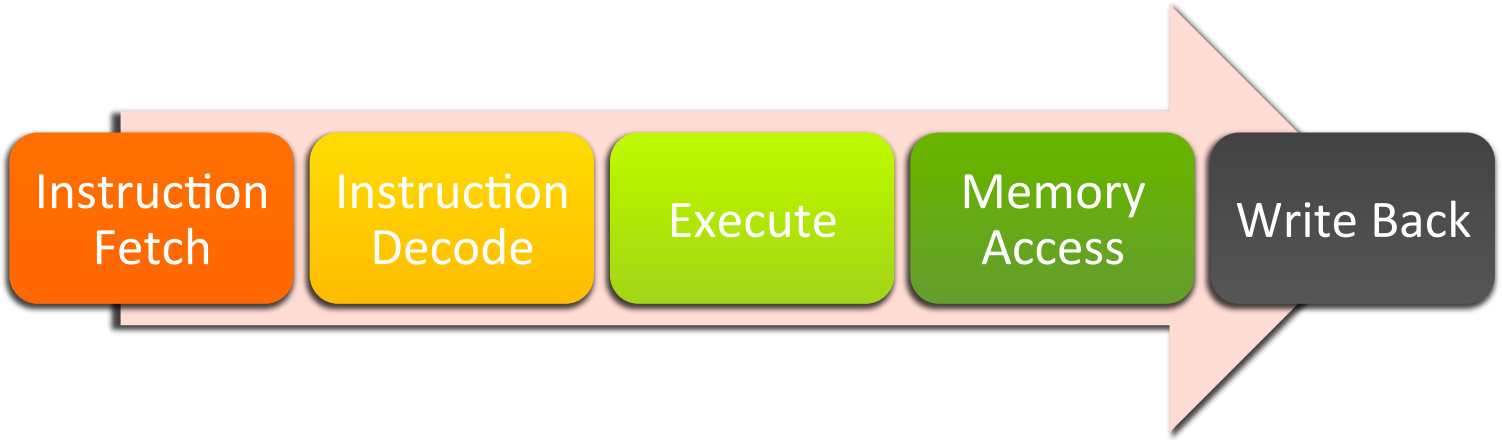
Assignments Due

- **Wednesday Apr 24th**
  - **Lab 12 due by 11:59pm**

MIPS Instruction Cycle

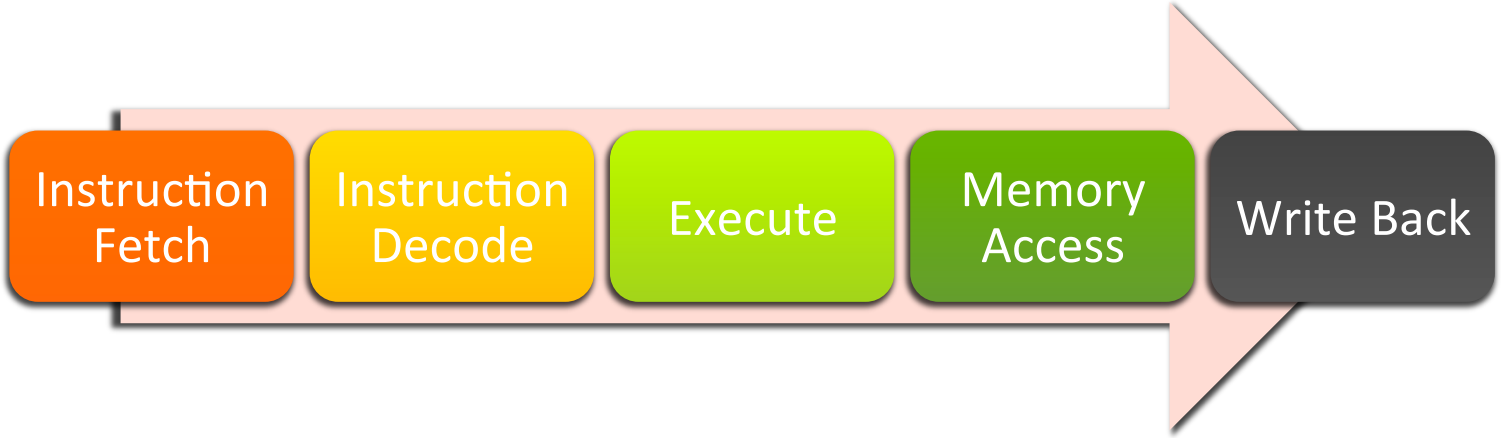
MIPS Instruction Cycle

- **How does the hardware MIPS processor execute a single instruction?**
- **With a 5-stage instruction cycle**



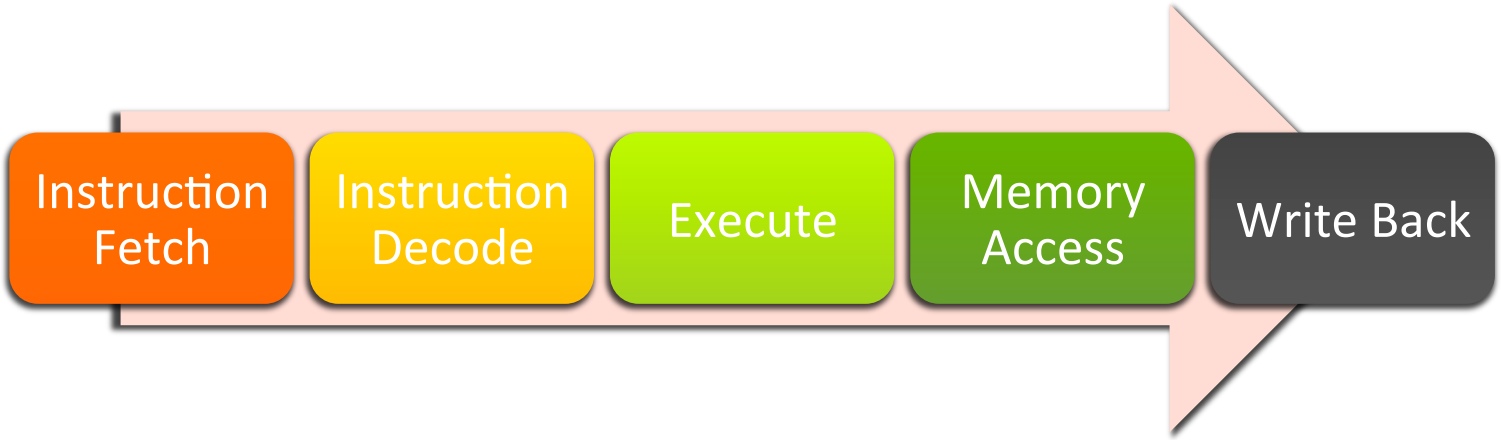
MIPS Instruction Cycle

- **Step 1 – Instruction Fetch (IF)**
  - Retrieve next instruction from memory *(check the instruction cache first!)*
  - **Program Counter** (PC) register stores address of next instruction to be retrieved/executed



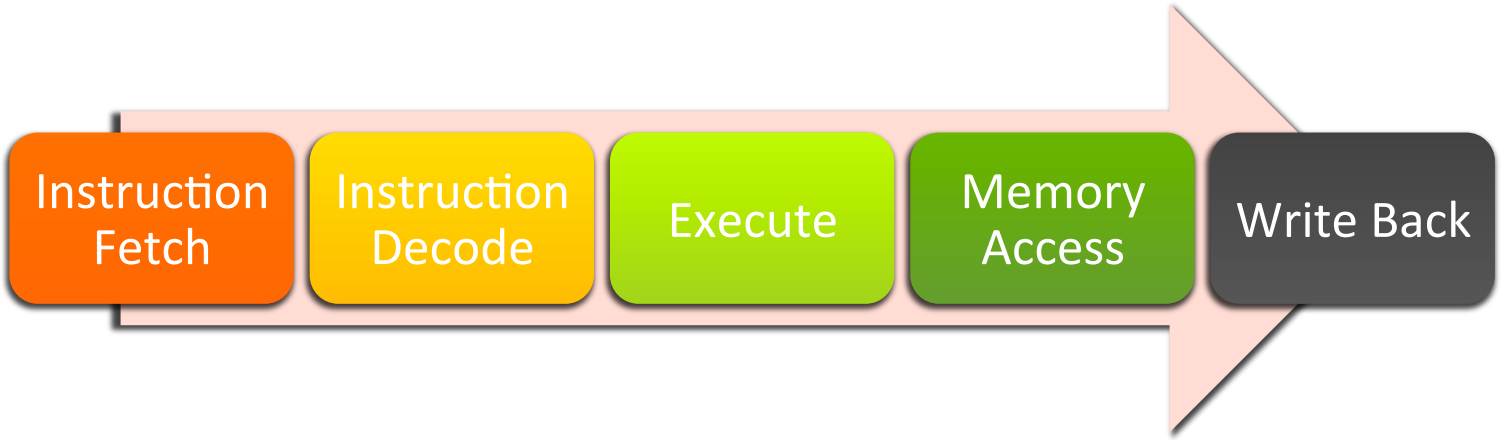
MIPS Instruction Cycle

- **Step 2 – Instruction Decode (ID)**
  - Decode instruction – what should we do?
  - Retrieve input values from registers



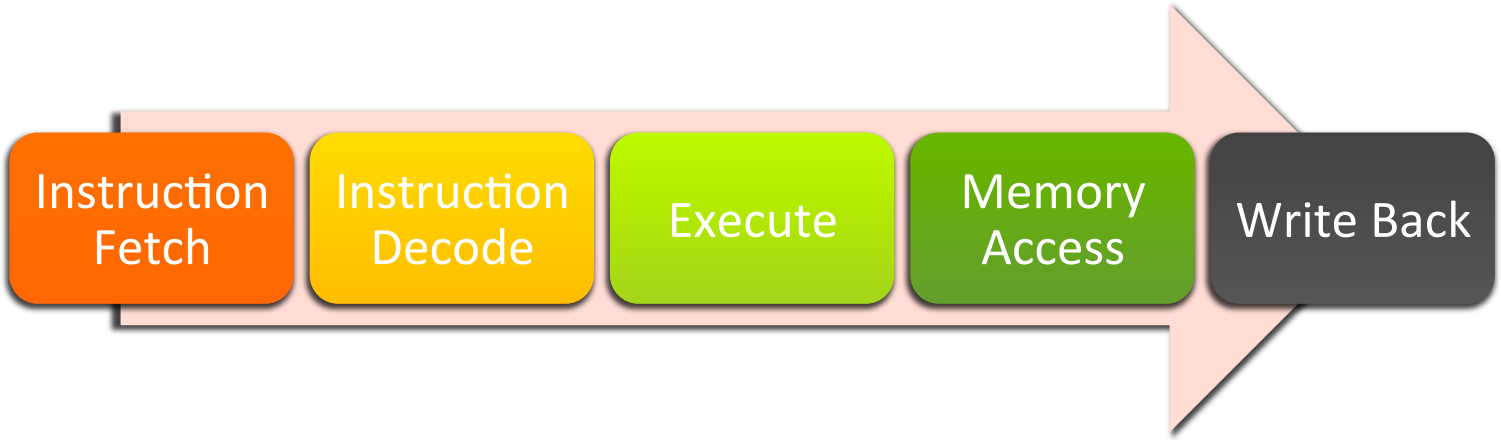
MIPS Instruction Cycle

- **Step 3 – Execute (EX)**
  - ALU performs arithmetic or logical operation
  - Operation might be calculating a memory address



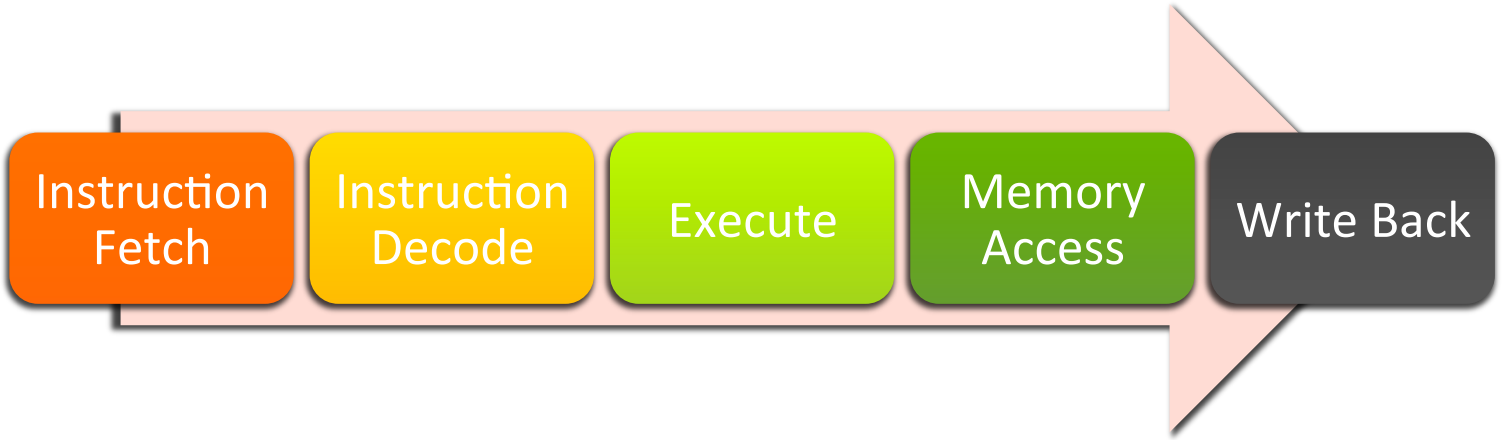
MIPS Instruction Cycle

- **Step 4 – Memory Access (MEM)**
  - Read/write memory *if necessary*
    (Check the data cache first!)



MIPS Instruction Cycle

- **Step 5 – Write Back (WB)**
  - Write final result of instruction to register *if necessary*



Example 1 – ADD $s0,$s1,$s2

1. **IF**: Load instruction from memory; increment PC
2. **ID**: Determine operation is “add”; Load $s1 and $s2 from registers
3. **EX**: ALU performs addition operation
4. **MEM**: No operation (no-op)
5. **WB**: Output of ALU written to $s0

Example 2 – LW $s0,10($t1)

1. **IF**: Load instruction from memory, increment PC
2. **ID**: Determine operation is "load word"; retrieve $t1 from register
3. **EX**: ALU calculates memory address of desired data ($t1 plus 10 sign-extended to full 32 bits)
4. **MEM**: Retrieve data from memory at address calculated by ALU *(check the data cache first!)*
5. **WB**: Output of memory written to $s0

Example 3 – SW $s0,20($t1)

1. **IF**: Load instruction from memory, increment PC
2. **ID**: Determine operation is “store word”; retrieve $s0 and $t1 from registers
3. **EX**: ALU calculates memory address of storage location ($t1 plus 20 sign-extended to full 32 bits)
4. **MEM**: Store value from $s0 to memory at address calculated by ALU *(write goes to the data cache!)*
5. **WB**: No operation (no-op)

Example 4 – BEQ $t1,$t2,label

1. **IF**: Load instruction from memory, increment PC
2. **ID**: Determine operation is "branch on equal"; retrieve $t1 and $t2 from registers
3. **EX**: ALU calculates memory address of location to jump to *if* the comparison is true (PC + label sign-extended to full 32 bits); ALU also compares $t1 and $t2 for equality
4. **MEM**: If comparison is equal, PC = address calculated by ALU. Otherwise, PC is unchanged
5. **WB**: No operation (no-op)

Pipelining

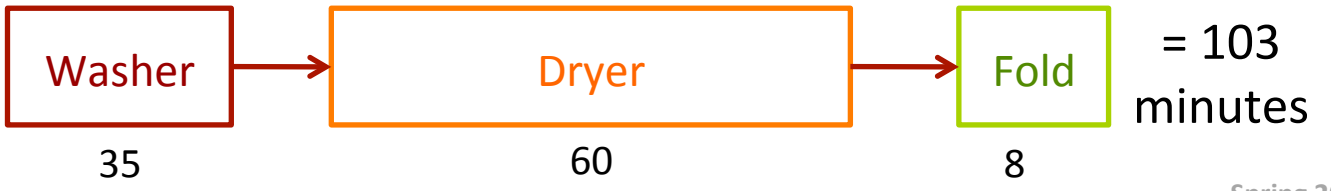
Instruction Cycle

The performance of our 5-step instruction cycle is slow if we only do one instruction at a time

New Goal: Run the instruction cycle quickly / efficiently

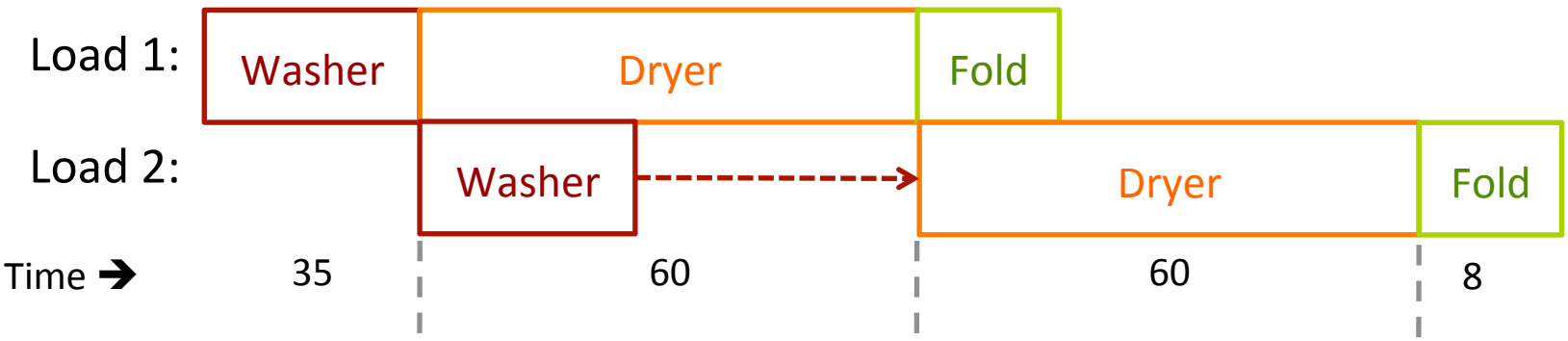
Instruction Cycle

- A laundry analogy…
  - Laundry cycle instead of instruction cycle
- Doing laundry in your residence hall
  - Washing machine – 35 minutes
  - Dryer – 60 minutes
  - Folding / Hanging – 8 minutes
- How do you do one load of laundry the fastest?



Instruction Cycle for Laundry

- **How do you do two loads of laundry the fastest?**
  - Back to back?
    - 206 minutes total
    - Leaves machines idle at different times
  - Concurrently?



Total: 163 minutes

Pipelining

- This is **pipelining**
  - Performing work in parallel instead of sequentially
    - Goal: Keep all hardware busy
  - Provides for **instruction level parallelism (ILP)**
    - Executing more than one instruction at a time

Without Pipelining:

Instr. #	Stage							
1	IF	ID	EX	MEM	WB	*Finish first instruction…*		
2				*… before starting second*		IF	ID	EX
3								
Cycle	1	2	3	4	5	6	7	8

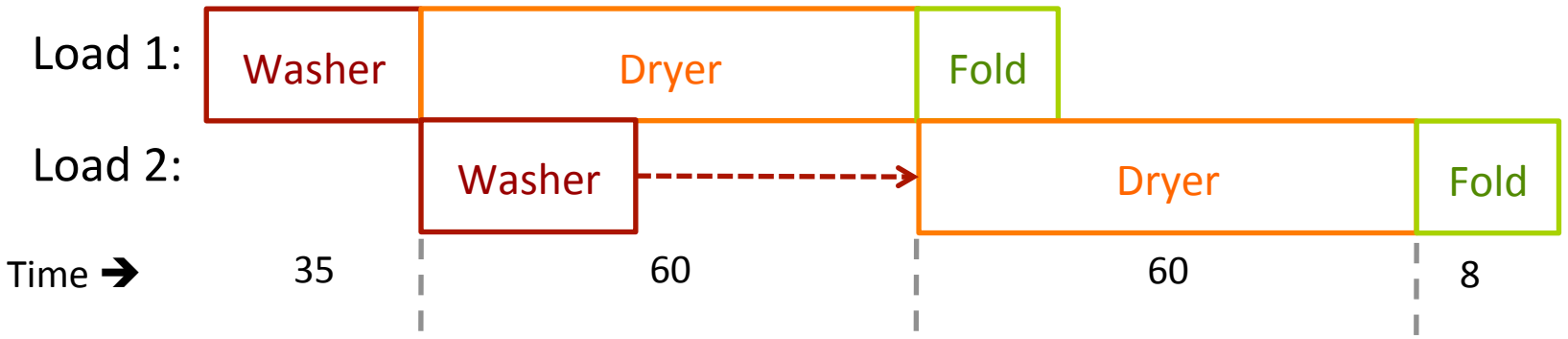
With Pipelining:

Instr. #	Pipeline Stage					
1	IF	ID	EX	MEM	WB	
2		IF	ID	EX	MEM	WB
3			IF	ID	EX	MEM
Cycle	1	2	3	4	5	6

Deeper Pipelining

- **We can do better than this**
- (Original) Laundry Room Specifications:
  - Washing machine – 35 minutes
  - Dryer – 60 minutes
  - Folding / Hanging – 8 minutes
- **What is the bottleneck in our simple pipeline?**
  - Drying takes much longer than the other stages
  - This slows down the entire laundry process

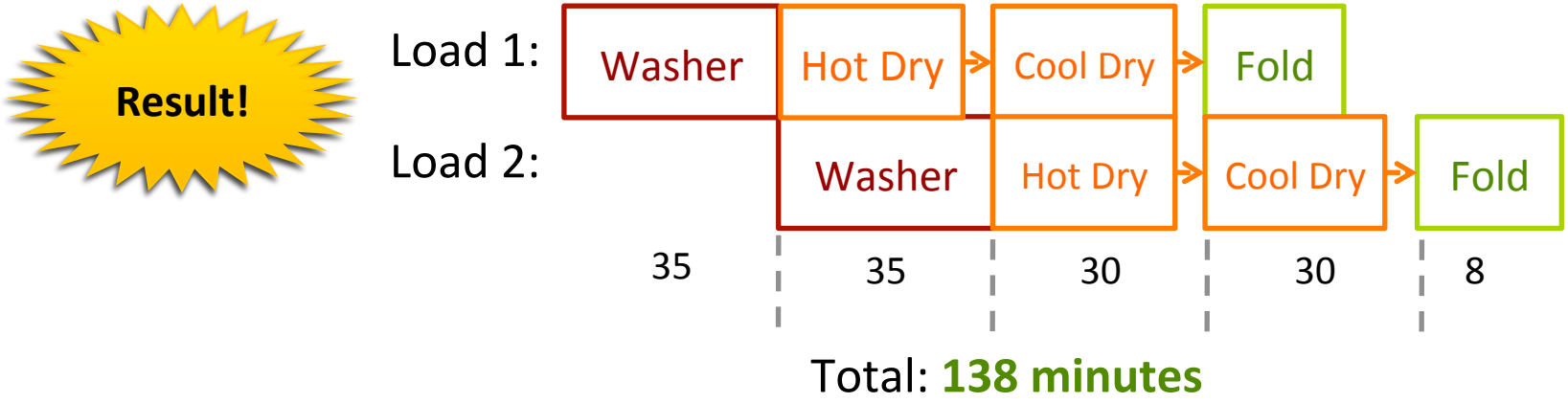
Pipelining / Laundry Revisited



Total: 163 minutes

- How can we fix it? Get two dryers
  - Operate them in parallel, or …
  - Operate them in series for half the time
    - Each has a specialized task
    - First dryer set to hot (initial drying)
    - Second dryer set to cool (final drying / prevent shrinking)

Pipelining / Laundry Revisited



- How can we fix it? Get two dryers
  - Operate them in parallel, or …
  - Operate them in series for half the time
    - Each has a specialized task
    - First dryer set to hot (initial drying)
    - Second dryer set to cool (final drying / prevent shrinking)

Pipelining / Laundry Revisited

- Better performance
  - 206 minutes → 163 minutes → **138 minutes**
  - But now we're limited by the washer speed
- How do we fix this?
  - Buy more machines, each doing smaller parts of the task
- **Could I benefit from 10 machines? 100? 1000?**
  - Not shown in timeline: Time required to advance laundry from one stage to the next
  - The time spent moving laundry between machines could exceed the time spent in the machines ☹
  - System becomes increasingly complex to design ☹

Pipeline Challenge 1

- **Ideal** pipeline speedup is equal to pipeline depth
  - 5 stages? Program could run at best 5 times faster
- **Pipeline challenge – only achieve ideal speedup if the pipeline is perfectly balanced**
  - The hardware in every stage takes the exact same amount of time to operate
- Most pipelines are not balanced
  - Example: loading data from memory is slower than decoding instruction
- **Do we set processor frequency to *fastest* or *slowest* stage?**
  - **Slowest stage** – otherwise it won't have time to finish

Pipeline Challenge 2

- **Problem: We might not always be able to keep the pipeline full of instructions**
- **Hazards** cause pipeline conflicts and stalls
  - Data hazards (dependencies)
  - Structural hazards (resource conflicts)
  - Control hazards (conditional branching)

Data Hazard

Program correctness depends on executing instructions in original order

Read After Write

```
add $s1,$t1,$t2
add $s2,$t3,$t4
add $t4,$s1,$s2
```

Third add cannot proceed until first two are complete!

Write After Read

```
add $t1,$s1,$t2
add $s1,$t3,$t4
```

Second add cannot write result until after first add has read its inputs!

Write After Write

```
add $s1,$t1,$t2
add $s1,$t3,$t4
```

Second add cannot write result until after first add has written its result!

Structural Hazard, Control Hazard

- **Structural hazard**
  - Part of the processor hardware is required by two different instructions at the same time
  - Example: A shared memory, shared ALU, shared data bus, etc…
- **Control hazard**
  - The processor needs to know which instruction will be executed next, and it can't until the branch is determined

Instruction-Level Pipelining

- Hazards can cause pipeline to **stall** or **flush**
  - **Stall** – pipeline is delayed for a cycle
  - **Flush** – all instructions in pipeline are deleted
- Clever hardware or clever assembly programmers (or *optimizing* compilers) can reduce the effects of these hazards
  - But not fully eliminate them…

Intel Pipelining

- Almost all Intel chips (286, 386, 486, etc…) have some degree of pipelining
- Pipelining was first seriously applied to the **Intel 486** chip in 1989
  - Could complete an ALU instruction (coming from a register, going to a register) *every clock cycle*
- Pipelining got better with the **Pentium** chip in 1993
  - Double-wide: *Two instructions* are sent down the pipeline every cycle! (Requires two ALUs, etc…)

Intel Pipelining

- Pipeline depth changed over time:
  - Original Pentium: 5 stages
  - Pentium 2: 12 stages
  - Pentium 3: 14 stages
  - Pentium 4: 20-24 stages
  - Pentium 4 extreme edition: 31 stages
  - **Why were the pipelines getting longer?**
- Today
  - Core i7 has a 17-stage pipeline

MIPS Pipelining

- Like Intel, the pipeline size of the MIPS processors has grown
  - R2000 and R3000 have 5-stage pipelines
  - R4000 and R4400 have 8-stage pipelines
  - R10000 has three pipelines:
    - 5-stage pipeline for integer instructions
    - 7-stage pipeline for floating-point instructions
    - 6-state pipeline for LOAD/STORE instructions

Parallelism

Instruction-Level Parallelism

- Example program: *(imagine it was in assembly)*

```
①   e = a + b;
②   f = c + d;
③   g = e * h;
```

- Assume we have a processor with "lots" of ALUs
  - **What instructions can be executed in parallel?**
  - **What instructions cannot be executed in parallel?**

Instruction-Level Parallelism

- Example program 2: *(imagine it was in assembly)*

```
①   e = a + b;
②   f = c + d;
③   if(e > f)
④     a = 15;
⑤   else
⑥     a = 18;
⑦   g = h + 30;
```

- Assume we have a processor with "lots" of ALUs
  - **What instructions can be executed in parallel?**
  - **What instructions cannot be executed in parallel?**
    - ***If we tried really hard, could we run them in parallel?***

Instruction-Level Parallelism

- This is **instruction-level parallelism**
  - Finding instructions in the *same* program that can be executed in parallel
  - **Different** from multi-core parallelism, which executes instructions from *different* programs in parallel
- You can do this in a single "core" of a CPU
  - Adding more ALUs to the chip is easy
  - Finding the parallelism to exploit is harder…
  - Getting the data to the ALUs is harder…

Instruction-Level Parallelism

- **Instruction-level parallelism is good** ☺
  - Let's find as much of it as possible and use it to decrease execution time!
- Two competing methods:
  - **Superscalar**: the *hardware* finds the parallelism
  - **VLIW**: the *compiler* finds the parallelism
- Both designs have **multiple execution units** (e.g. ALUs) in a **single** processor core

MIMD – Superscalar

- **Superscalar** designs – the *hardware* finds the **instruction-level parallelism** while the program is running
- Challenges
  - CPU *instruction fetch unit* must simultaneously retrieve several instructions from memory
  - CPU *instruction decoding unit* determines which of these instructions can be executed in parallel and combines them accordingly
    - **Complicated!**

MIMD – VLIW

- **Very long instruction word (VLIW)** designs – the *compiler* finds the **instruction-level parallelism** before the program executes
  - The *compiler* packs multiple instructions into one **long** instructions that the hardware executes in parallel
- Arguments:
  - **For**: Simplifies hardware, plus the compiler can better identify instruction dependencies (it has more time to work)
  - **Against**: Compilers cannot have a view of the run time code, and must plan for all possible branches and code paths
- Examples: Intel Itanium, ATI R600-R900 GPUs

Instruction-Level Parallelism

- Back to the example program:

```
①   e = a + b;
②   f = c + d;
③   if(e > f)
④     a = 15;
⑤   else
⑥     a = 18;
⑦   g = h + 30;
```

- More techniques for ILP
- **Speculative execution** (or **branch prediction**)
  - Guess that e>f, and execute line 4 immediately…
- **Out-of-order execution**
  - Execute line 7 before 4-6, since it doesn't depend on them