CS 61C: Great Ideas in Computer Architecture
Instruction Level Parallelism: Multiple Instruction Issue
Instructor: David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

You Are Here!

• Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
• Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
• Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
• Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
• Hardware descriptions
  All gates @ one time
• Programming Languages

The BIG Picture

• Pipelining improves performance by increasing instruction throughput: exploits ILP
  — Executes multiple instructions in parallel
  — Each instruction has the same latency
• Subject to hazards
  — Structure, data, control
• Stalls reduce performance
  — But are required to get correct results
• Compiler can arrange code to avoid hazards and stalls
  — Requires knowledge of the pipeline structure

 Agenda

• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Example AMD Barcelona
• Intel Nehalem
• Big Picture: Types of Parallelism
• Peer Instruction Questions (as time permits)
• Summary

Greater Instruction-Level Parallelism (ILP)

1. Deeper pipeline (5 => 10 => 15 stages)
   – Less work per stage ⇒ shorter clock cycle
2. Multiple issue superscalar
   – Replicate pipeline stages ⇒ multiple pipelines
   – Start multiple instructions per clock cycle
3. CPI < 1, so can use Instructions Per Cycle (IPC)
   – E.g., 4 GHz 4-way multiple-issue
     • 16 BIPS, peak CPI = 0.25, peak IPC = 4
     • But dependencies reduce this in practice

Multiple Issue

• Static multiple issue
  — Compiler groups instructions to be issued together
  — Packages them into "issue slots"
  — Compiler detects and avoids hazards
• Dynamic multiple issue
  — CPU examines instruction stream and chooses instructions
to issue each cycle
  — Compiler can help by reordering instructions
  — CPU resolves hazards using advanced techniques at runtime
Superscalar Laundry: Parallel per stage

- Task:
  - Light clothing
  - Dark clothing
  - Very dirty clothing

- More resources, HW to match mix of parallel tasks?

Pipeline Depth and Issue Width

• Intel Processors over Time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>80x86</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2000 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>3000 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3400 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>

StaVc_multiple_issue

• Compiler groups instructions into **issue packets**
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
  - Think of an issue packet as a very long instruction
    - Specifies multiple concurrent operations
    - Called VLIW for Very Long Instruction Word

Symmetric multiprocessor

- No dependencies with a packet
- Possibly some dependencies between packets
  - Varies between ISAs; compiler must know!
- Pad with nop if necessary

Scheduling Static Multiple Issue

MIPS with Static Dual Issue

- Dual-issue packets
  - One ALU/branch instruction + One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WR</td>
</tr>
<tr>
<td>1 + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WR</td>
</tr>
<tr>
<td>1 + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WR</td>
</tr>
<tr>
<td>1 + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WR</td>
</tr>
<tr>
<td>1 + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WR</td>
</tr>
<tr>
<td>1 + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WR</td>
</tr>
</tbody>
</table>
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result for load/store in same packet
    - add $t0, $s0, $s1
      - load $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS
  
  Loop:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALU/branch</th>
<th>Load/store</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1, $t0, $s1</td>
<td>add $t0, $t0, $s2</td>
<td>lw $t0, 0($s1)</td>
</tr>
<tr>
<td>add $t1, $t1, $s1</td>
<td>lw $t1, 12($s1)</td>
<td></td>
</tr>
<tr>
<td>lw $t2, 0($s1)</td>
<td>lw $t2, 0($s1)</td>
<td></td>
</tr>
<tr>
<td>lw $t3, 4($s1)</td>
<td>lw $t3, 4($s1)</td>
<td></td>
</tr>
<tr>
<td>lw $t4, 4($s1)</td>
<td>lw $t4, 4($s1)</td>
<td></td>
</tr>
<tr>
<td>sw $t0, 0($s1)</td>
<td>sw $t1, 12($s1)</td>
<td></td>
</tr>
<tr>
<td>sw $t2, 0($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t3, 4($s1)</td>
<td>sw $t4, 4($s1)</td>
<td></td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- IPC = 5/4 = 1.25 (vs. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called register renaming
  - Avoid loop-carried anti-dependencies
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name but no real dependency between instructions

Loop Unrolling Example

- Project 4: Pipelined Cycle Processor in Logicsim
  - due 4/15 (No Part 1)
- Extra Credit: Fastest Version of Project 3
  - Due 4/22 11:59 PM
- Final Review: April 29, 2PM-5PM, 2050 VLSB
- All grades finalized: 4/27
- Final: Wednesday May 9 11:30-2:30 (TBD)

Administrivia

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of more registers and bigger code
61C in the News
“A Rose-Colored View May Come Standard”, NY Times, 4/5/12

- The glasses, which are still in a prototype stage, would place a small see-through display screen above a person’s eye that can show maps and other data. The wearer could use voice commands to, say, pull up directions or send a message to a friend.

Dynamic Multiple Issue
- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... instructions each cycle
  - Avoiding structural and data hazards
- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling
- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example
  - lw $t0, 20($s2)
  - addu $t1, $t0, $t2
  - subu $s4, $s4, $t3
  - slti $t5, $s4, 20
  - Can start subu while addu is waiting for lw
  - Especially if cache misses, can execute many instructions

Why Do Dynamic Scheduling?
- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Agenda
- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- Example AMD Barcelona
- Intel Nehalem
- Big Picture: Types of Parallelism
- Peer Instruction Questions (as time permits)
- Summary

Speculation
- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll back and do the right thing
- Examples
  - Speculate on branch outcome (Branch Prediction)
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
  - Can be done in hardware or by compiler
  - Common to static and dynamic multiple issue
Pipeline Hazard:
Matching socks in later load

Not a simple linear pipeline
- 3 major units operating in parallel
- 1 Instruction fetch and issue unit
  - Issues instructions in program order
- Many parallel functional (execution) units
  - Each functional unit has input buffers called Reservation Stations
  - Holds operands and records the operation
  - Can execute instructions out of program order (OOO)
- 1 Commit unit
  - Gets results from functional unit and saves in buffers called Reorder Buffer
  - Stores results once branch resolved so OK to execute
  - Commits results in program order

Out-of-Order Laundry: Don’t Wait

• A depends on D; stall since folder tied up;

Out-of-Order Execution (1/2)
- Basically, unroll loops in hardware
  1. Fetch instructions in program order (≤4/clock)
  2. Predict branches as taken/untaken
  3. To avoid hazards on registers, rename registers using a set of internal registers (~80 registers)
  4. Collection of renamed instructions might execute in a window (~60 instructions)
  5. Execute instructions with ready operands in 1 of multiple functional units (ALUs, FPUs, Ld/St)

Out-of-Order Execution (2/2)
- Basically, unroll loops in hardware
  6. Buffer results of executed instructions until predicted branches are resolved in reorder buffer
  7. If predicted branch correctly, commit results in program order
  8. If predicted branch incorrectly, discard all dependent results and start with correct PC

Dynamically Scheduled CPU
Out Of Order Intel

- All use OOO since 2001

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order</th>
<th>Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>C86</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>75W</td>
<td></td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>3</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
<td></td>
</tr>
<tr>
<td>PA Inhberdele</td>
<td>2001</td>
<td>3000MHz</td>
<td>3</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>100W</td>
<td></td>
</tr>
<tr>
<td>PA Prescott</td>
<td>2004</td>
<td>3000MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
<td></td>
</tr>
<tr>
<td>Core</td>
<td>2005</td>
<td>2000MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
<td></td>
</tr>
<tr>
<td>Core i7 Yorkfield</td>
<td>2008</td>
<td>2660 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>8</td>
<td>90W</td>
<td></td>
</tr>
<tr>
<td>Core i7 Guadalupe</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>8</td>
<td>130W</td>
<td></td>
</tr>
</tbody>
</table>

Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
  - Example AMD Barcelona
  - Example Intel Nehalem (lab computer)
- Big Picture: Types of Parallelism
- Peer Instruction Questions (as time permits)
- Summary

AMD Opteron X4 Microarchitecture

72 physical registers

AMD Opteron X4 Pipeline Flow

- For integer operations
- 12 stages (Floating Point is 17 stages)
- Up to 106 RISC-ops in progress

Intel Nehalem

- Look at microprocessor from Intel in servers in your 61C labs and your laptops
- Nehalem: code name for microarchitecture at heart of Core i7, Xeon 5500 series server chips
  - Intel legal said had to pick names of rivers for code names (vs. "AMD Destroyer")
- First released at end of 2008
- Die size 263 mm² at 45 nm, 731M transistors
- Intel Nehalem is 16 stages for integer operations
  - Intel calls RISC operations "Micro operations" or "μops"

Nehalem River, Oregon
Nehalem System Example:
Apple Mac Pro Desktop 2010

- Two Nehalem chips ("sockets"), each containing four processors ("cores") running at up to 2.93GHz
- Each chip has three DRAM channels attached, each 8 bytes wide at 1.066GHz (3*8.5GB/s)
- Can have up to two DIMMs on each channel (up to 4GB/DIMM)

Disk drives attached with 3Gb/s serial ATA link

QuickPath point-to-point system interconnect between CPUs and I/O. Up to 25.6GB/s per link.

PCI Express connections for graphics cards and other extension boards. Up to 8 GB/s per slot.

Slower peripherals (Ethernet, USB, Firewire, WiFi, Bluetooth, Audio)

Nehalem 12-inch Wafer: 280 dies

- Same diameter as medium Domino's pizza

Core Area Breakdown

Front-End Instruction Fetch & Decode

μOP is Intel name for internal RISC-like (MIPS) instruction, into which x86 instructions are translated

Loop Stream Detector (can run short loops out of the buffer)
x86 Decoding

- Translate up to 4 x86 instructions into μOPS (=MIPS or RISC instructions) each cycle
- Only first x86 instruction in group can be complex (maps to 1-4 μOPS), rest must be simple (map to one μOP)
- Even more complex instructions, jump into microcode engine which spits out stream of μOPS

Branch Prediction

- Part of instruction fetch unit
- Several different types of branch predictor
  - Details not public
- Two-level Branch Table Buffer
- Loop count predictor
  - How many backwards taken branches before loop exit
- Return Stack Buffer
  - Holds subroutine targets
  - Separate return stack buffer for each SMT thread

Split x86 in small μOPs, then fuse back into bigger units

Out-of-Order Execution Engine

- 20 Entry jmp Buffer
  - 4 ways
  - Regular Alias Table and Allocator
- 128 Entry Rootset Buffer (RCB)
  - 4 ways
  - 38 Entry Reservation Station
  - 7 ways
  - ALU
  - FLUSH
  - STORE
  - LOAD
  - STORE
  - FADD
  - FADD

Loop Stream Detectors save Power

- Branch prediction
- Fetch
- Decode

Intel® Core™2 Loop Stream Detector

- Branch prediction
- Fetch
- Decode
- 0 Instructions

Intel Core Microarchitecture (Nehalem) Loop Stream Detector

- Branch prediction
- Fetch
- Decode
- Retired Instructions

Extending Performance and Energy Efficiency
- Intel® SSE4.2 Instruction Set Architecture (ISA) Leadership in 2008

- SSE4 Instruction Set
  - Accelerated Sorting and Fast Processing
  - Accelerated Searching & Failing Recognition of Large Data Sets
  - New Communications Capabilities

- What should the applications, OS and VMM vendors do?:
  Understand the benefits & take advantage of new instructions in 2008. Provide us feedback on instructions. ETU was looking to the next generation of applications
**Intel® Hyper-Threading Technology**

- Also known as Simultaneous Multi-Threading (SMT)
  - Run 2 threads at the same time per core
- Take advantage of 4-wide execution engine
  - Keep it fed with multiple threads
- Hide latency of a single thread
- Most power efficient performance feature
  - Very low die area cost
  - Can provide significant performance benefit depending on application
  - Much more efficient than adding an entire core
- Intel® Core™ microarchitecture (Nehalem) advantages
  - Larger caches
  - Massive memory BW

**Multithreading effects in Out-of-Order Execution Core**

- Reorder buffer (remembers program order and exception status for in-order commit) has 128 entries divided statically and equally between both threads
- Reservation stations (instructions waiting for operands for execution) have 36 entries competitively shared by threads

**Vtune Performance Counters for Nehalem**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST_RETIRED.ANY</td>
<td>X86 instructions completed</td>
</tr>
<tr>
<td>UOPS_ISSUED.ANY</td>
<td>Micro-operations started</td>
</tr>
<tr>
<td>UOPS_ISSUED.STALL_CYCLES</td>
<td>Clock cycles no micro-operations were started</td>
</tr>
<tr>
<td>UOPS_RETIRED.ANY</td>
<td>Micro-operations completed</td>
</tr>
<tr>
<td>RESOURCE_STALLS.ANY</td>
<td>Pipeline stalls due to any hardware being busy</td>
</tr>
<tr>
<td>RESOURCE_STALLS.LOAD</td>
<td>Pipeline stalls due to load buffer being busy</td>
</tr>
<tr>
<td>RESOURCE_STALLS.STORE</td>
<td>Pipeline stalls due to store buffer being busy</td>
</tr>
<tr>
<td>RESOURCE_STALLS.RES_FULL</td>
<td>Pipeline stalls due to full Reservation Stations</td>
</tr>
<tr>
<td>RESOURCE_STALLS.RS_FULL</td>
<td>Pipeline stalls due to full ReOrder Buffer</td>
</tr>
<tr>
<td>UOPS_EXECUTED.CORE_STALL_CYCLES</td>
<td>Micro-operations completed per core</td>
</tr>
<tr>
<td>PEBs</td>
<td>Micro-operations completed per thread</td>
</tr>
</tbody>
</table>

**Managing Active Power**

- Operating system changes frequency as needed to meet performance needs, minimize power
  - Enhanced Intel SpeedStep® Technology
  - Referred to as processor P-States
- PCU tunes voltage for given frequency, operating conditions, and silicon characteristics

**Power Control Unit**

[Diagram showing the Power Control Unit with integrated power telemetry and P-States control, along with real-time sensors for temperature, current, power. Flexibility enables sophisticated algorithms, tuned for current operating conditions.]

**(Nehalem) Turbo Mode**

- Power Gating
  - Zero power for inactive cores
- Turbo Mode
  - In response to workload adds additional performance bits within headroom

**PCU automatically optimizes operating voltage**
**Does Multiple Issue Work?**

**The BIG Picture**

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

**Big Picture on Parallelism**

Two types of parallelism in *applications*

1. **Data-Level Parallelism (DLP):** arises because there are many data items that can be operated on at the same time
2. **Task-Level Parallelism (TLP):** arises because tasks of work are created that can operate largely in parallel

**Big Picture on Parallelism**

Hardware can exploit app Data LP and Task LP in 4 ways:

1. **Instruction-Level Parallelism:** Hardware exploits application DLP using ideas like pipelining and speculative execution
2. **SIMD architectures:** exploit app DLP by applying a single instruction to a collection of data in parallel
3. **Thread-Level Parallelism:** exploits either app DLP or TLP in a tightly-coupled hardware model that allows for interaction among parallel threads
4. **Request-Level Parallelism:** exploits parallelism among largely decoupled tasks and is specified by the programmer of the operating system

**Peer Question**

State if following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

<table>
<thead>
<tr>
<th></th>
<th>Super-</th>
<th>Out of</th>
<th>Specu-</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
<td>Order</td>
<td>lation</td>
<td>Renaming</td>
</tr>
<tr>
<td>Orange</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>Green</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Pink</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
</tr>
<tr>
<td>Yellow</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>

**Peer Question**

State if following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

<table>
<thead>
<tr>
<th></th>
<th>Super-</th>
<th>Out of</th>
<th>Specu-</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scalar</td>
<td>Order</td>
<td>lation</td>
<td>Renaming</td>
</tr>
<tr>
<td>Orange</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>Green</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Pink</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
</tr>
<tr>
<td>Yellow</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>
II. The number of pipe stages per instruction affects throughput, not

3. Instead of trying to make instructions take fewer cycles, we should

2. You cannot make ALU instructions take fewer cycles because of the

1. Allowing jumps, branches, and ALU instructions to take fewer stages

--longer pipelines are always a win (since less work per stage & a faster clock).

Not all instructions are active in every stage of the 5-stage pipeline.

Peer Question

--5/8/12

Instruct LP, SIMD, Thread LP, Request LP are examples of

--Parallelism above (\(\Lambda\)) the Instruction Set Architecture

--Parallelism explicitly at (\(\exists\)) the level of the ISA

--Parallelism below (\(\land\)) the level of the ISA

Peer Answer

Not all instructions are active in every stage of the 5-stage pipeline.

--Allowing jumps, branches, and ALU instructions to take fewer stages

--You cannot make ALU instructions take fewer cycles because of the

--Instead of trying to make instructions take fewer cycles, we should

--The number of pipe stages per instruction affects throughput, not latency.

Peer Instruction

I. Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

II. Longer pipelines are always a win (since less work per stage & a faster clock).

A) orange: I is True and II is True
B) green: I is False and II is True
C) pink: I is True and II is False
D) yellow: I is False and II is False

Peer Instruction

Inst. LP  SIMD  Thr. LP  Req. LP
Orange  \(\vee\)  =  =  \(\land\)
Green  =  =  \(\land\)  \(\land\)
Pink  \(\vee\)  =  \(\land\)  \(\land\)
Yellow  =  \(\land\)  \(\land\)  \(\land\)

Peer Answer

I. Thanks to pipelining, I have reduced the time it took me to wash my one shirt.

II. Longer pipelines are always a win (since less work per stage & a faster clock).

Orange: 1
Green: 2
Pink: 3
Yellow: 4
“And in Conclusion, …”

- Big Ideas of Instruction Level Parallelism
- Pipelining, Hazards, and Stalls
- Forwarding, Speculation to overcome Hazards
- Multiple issue to increase performance
  - IPC instead of CPI
- Dynamic Execution: Superscalar in-order issue, branch prediction, register renaming, out-of-order execution, in-order commit
  - “unroll loops in HW”, hide cache misses