EECS 470

Control Hazards and ILP

Lecture 3 – Fall 2024

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Announcements

- HW1 due Today @10pm
  - I'll be around after class to answer questions
- Project 1 due Tuesday 1/23.
  - Note: can submit multiple times, only last one counts.
    - Feedback is minimal.
    - New grader machine, could have issues.
- HW2 posted early next week.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Today

- Review and finish up control hazards. Examine other pipeline changes
- Costs and Power
- Instruction Level Parallelism (ILP) and Dynamic Execution

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Three approaches to handling data hazards

- Avoidance
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Problems with each solution

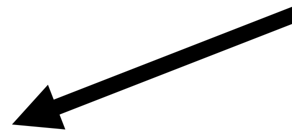
- Avoidance (static)
  - Predication
    - Needs larger instruction encodings
    - If the branch body is long, lots of noops
    - CMOV reduces encoding issues, but increases useless ops.
- Detect and Stall (dynamic)
  - Stall on every branch
  - Fair bit of hardware complexity
- Speculate and squash (dynamic)
  - Stall on every mispredicted branch
  - More hardware complexity.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Avoidance Via Predication

```
if (a == b) {
    x++;
    y = n / d;
}
```

```
sub     t1 ← a, b
jnz     t1, PC+2
add     x ← x, #1
div     y ← n, d
```



```
sub         t1 ← a, b
add(!t1)    x ← x, #1
div(!t1)    y ← n, d
```

```
sub         t1 ← a, b
add         t2 ← x, #1
div         t3 ← n, d
cmov(!t1)   x ← t2
cmov(!t1)   y ← t3
```

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Improving pipeline performance

- Add more stages
- Widen pipeline

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Adding pipeline stages

- Pipeline frontend
  - Fetch, Decode
- Pipeline middle
  - Execute
- Pipeline backend
  - Memory, Writeback

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Adding stages to fetch, decode

- Delays hazard detection
- No change in forwarding paths
- No performance penalty with respect to data hazards

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Adding stages to execute

- Check for structural hazards
  - ALU not pipelined
  - Multiple ALU ops completing at same time
- Data hazards may cause delays
  - If multicycle op hasn't computed data before the dependent instruction is ready to execute
- Performance penalty for each stall

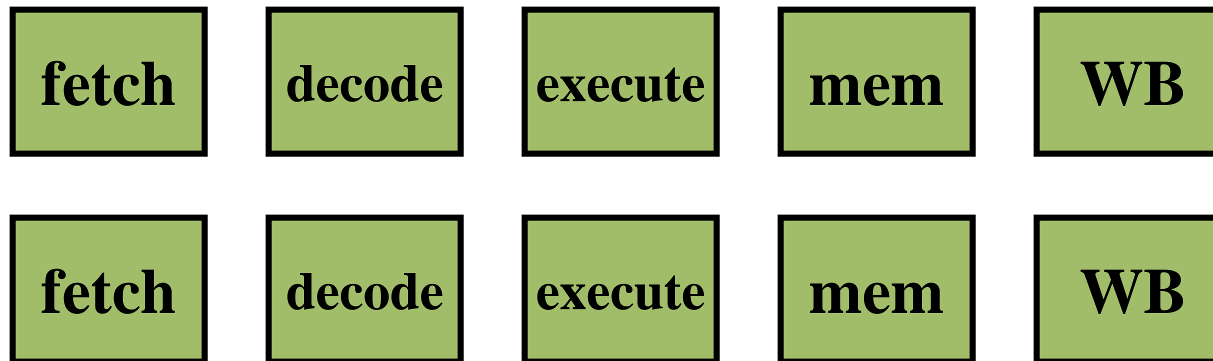
Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Adding stages to memory, writeback

- Instructions ready to execute may need to wait longer for multi-cycle memory stage
- Adds more pipeline registers
  - Thus more source registers to forward
    - More complex hazard detection
    - Wider muxes
    - More control bits to manage muxes

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Wider pipelines



More complex hazard detection

- **2X pipeline registers to forward from**
- **2X more instructions to check**
- **2X more destinations (muxes)**
- **Need to worry about dependent instructions in the same stage**

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Making forwarding explicit

- add r1 ← r2, EX/Mem ALU result
  - Include direct mux controls into the ISA
  - Hazard detection is now a compiler task
  - New micro-architecture leads to new ISA
    - Is this why this approach always seems to fail? (e.g., simple VLIW, Motorola 88k)
  - Can reduce some resources
    - Eliminates complex conflict checkers

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin,
Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Today

- Review and finish up control hazards. Examine other pipeline changes
- Costs and Power
- Instruction Level Parallelism (ILP) and Dynamic Execution

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Digital System Cost

Cost is also a key design constraint

- Architecture is about trade-offs
- Cost plays a major role

Huge difference between Cost & Price

E.g.,

- Higher Price → Lower Volume → Higher Cost → Higher Price
- Direct Cost
- List vs. Selling Price

Price also depends on the customer

- College student vs. US Government

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Direct Cost

Cost distribution for a Personal Computer

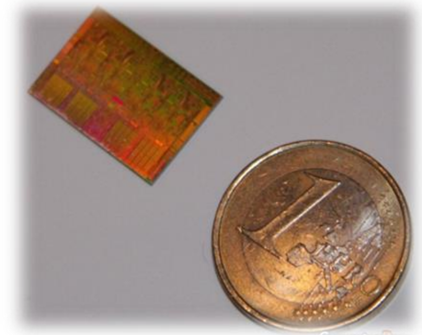
- Processor board 37%
  - CPU, memory,
- I/O devices 37%
  - Hard disk, DVD, monitor, …
- Software 20%
- Tower/cabinet 6%

Integrated systems account for a substantial fraction of cost

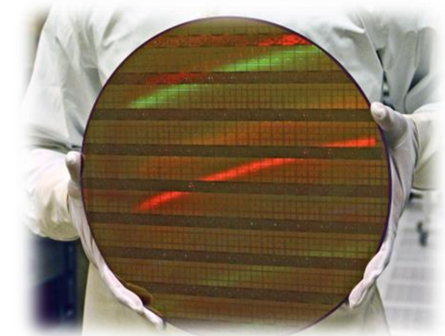
Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

IC Cost Equation

$$\text{IC cost} = \frac{\text{Die cost} + \text{Test cost} + \text{Packaging cost}}{\text{Final test yield}}$$



$$\text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies/wafer x Die yield}}$$

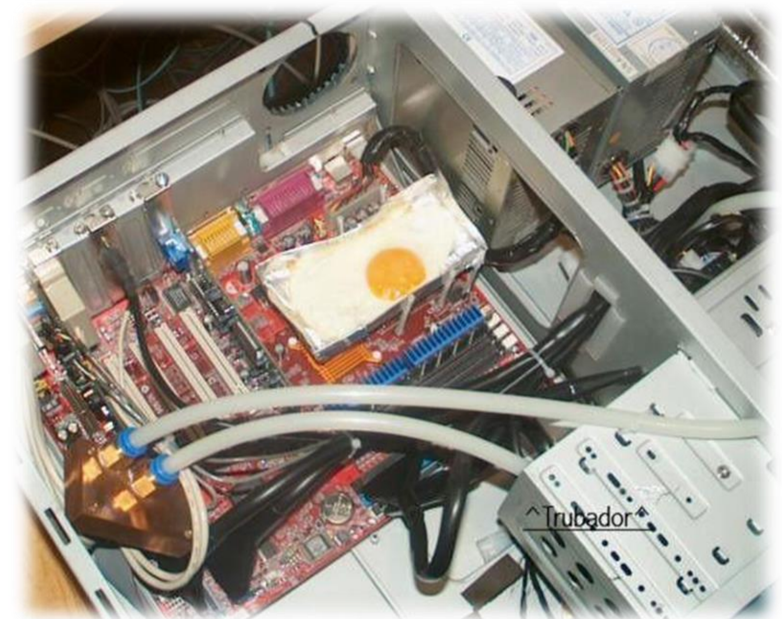


Die yield = f(defect density, die area)

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

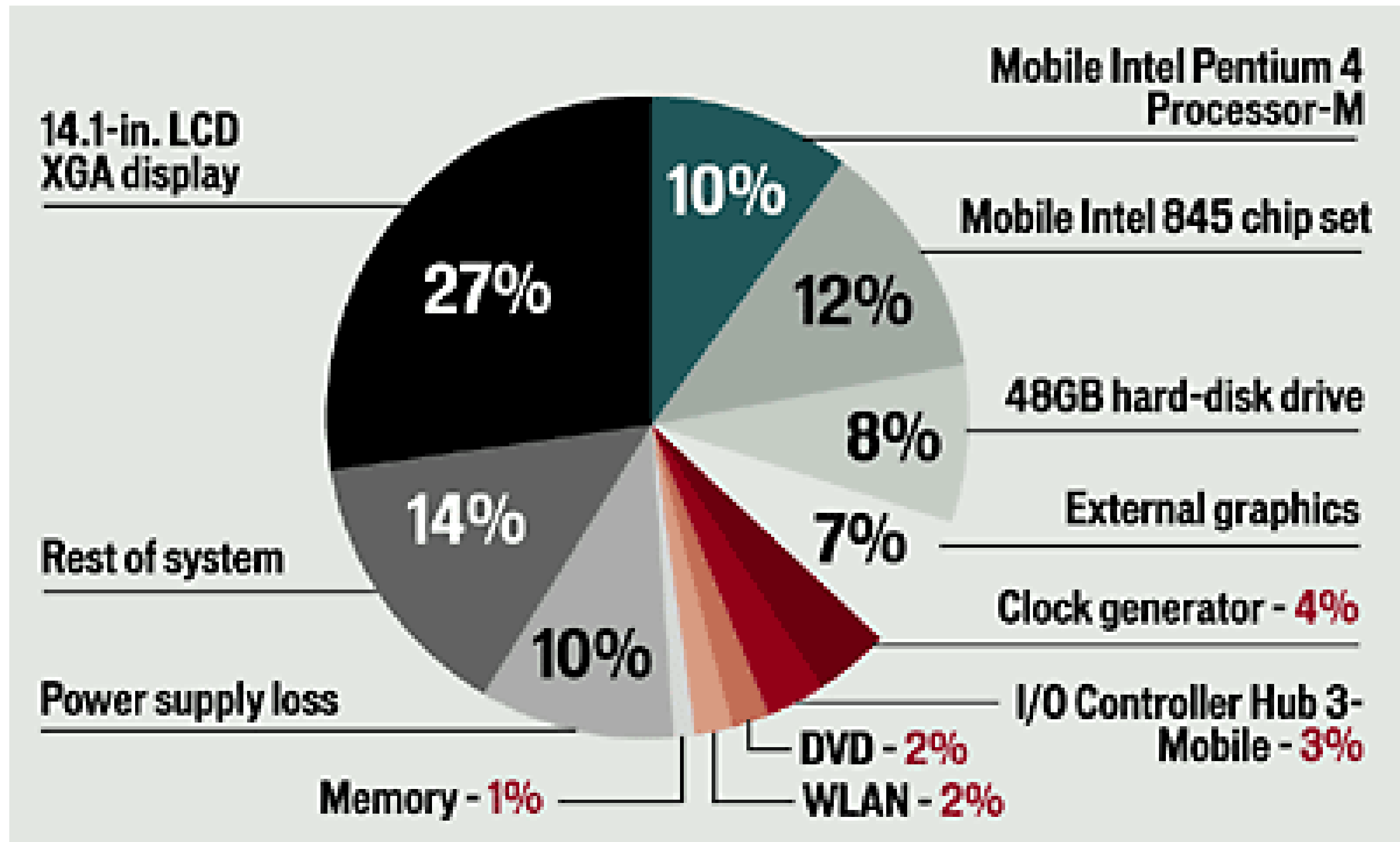
Why is power a problem in a μP?

- Power used by the μP, vs. system power
- Dissipating Heat
  - Melting (very bad)
  - Packaging (to cool → $)
  - Heat leads to poorer performance.
- Providing Power
  - Battery
  - Cost of electricity



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Where does the juice go in laptops?

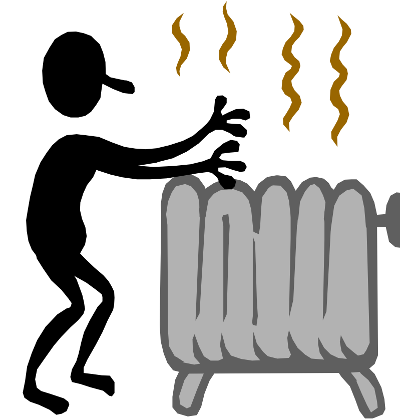


- Others have measured ~55% processor increase under max load in laptops

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

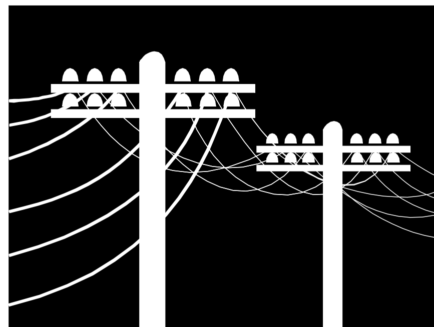
Why worry about power dissipation?

Battery life



Thermal issues: affect cooling, packaging, reliability, timing

Environment



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Power-Aware Computing Applications

Temperature/Current-Constrained

Energy-Constrained Computing

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Today

- Review and finish up other options
- Costs and Power
- Instruction Level Parallelism (ILP) and Dynamic Execution

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Limitations of Scalar Pipelines

Upper Bound on Scalar Pipeline Throughput

Limited by IPC=1 "Flynn Bottleneck"

Performance Lost Due to Rigid In-order Pipeline

Unnecessary stalls

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Terms

- **Instruction parallelism**
  - Number of instructions being worked on
- **Operation Latency**
  - The time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction. For example, if the result of an Add instruction can be used as an operand of an instruction that is issued in the cycle after the Add is issued, we say that the Add has an operation latency of one.
- **Peak IPC**
  - The maximum sustainable number of instructions that can be executed per clock.

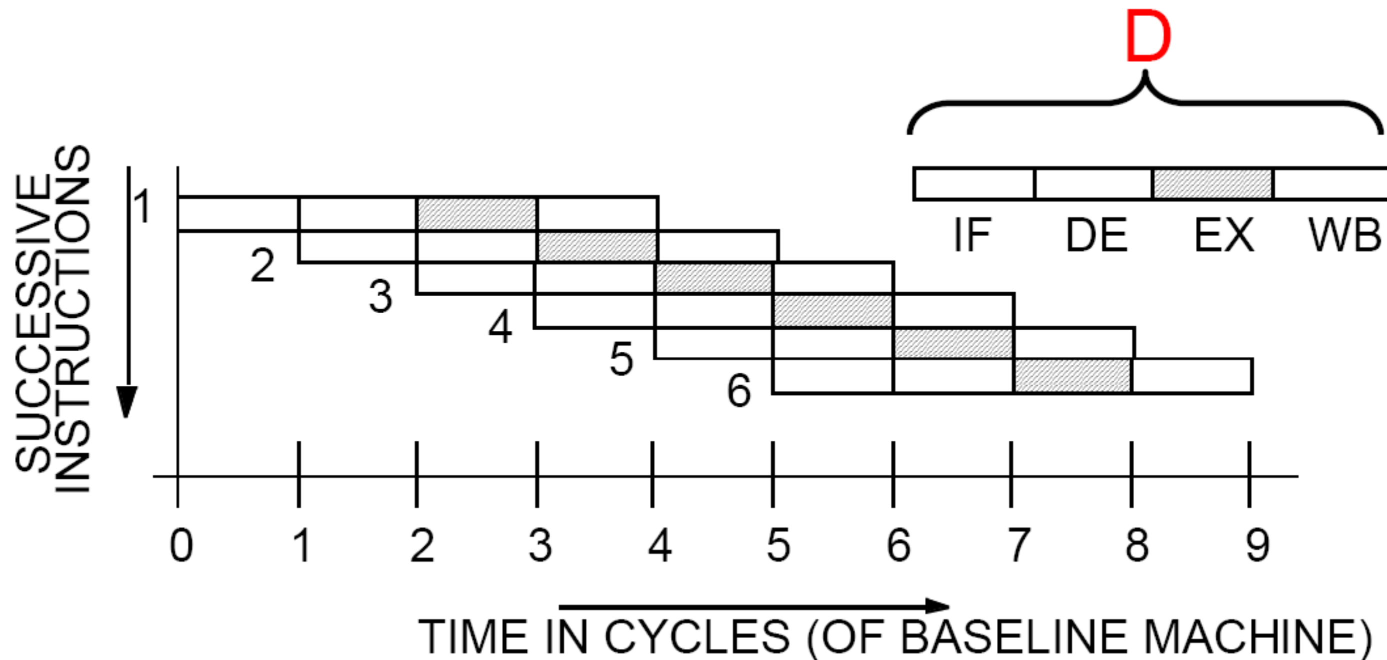
Performance modeling for computer architects, C. M. Krishna

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Architectures for Exploiting Instruction-Level Parallelism

Scalar Pipeline (baseline)

- Instruction Parallelism = D
- Operation Latency = 1
- Peak IPC = 1



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

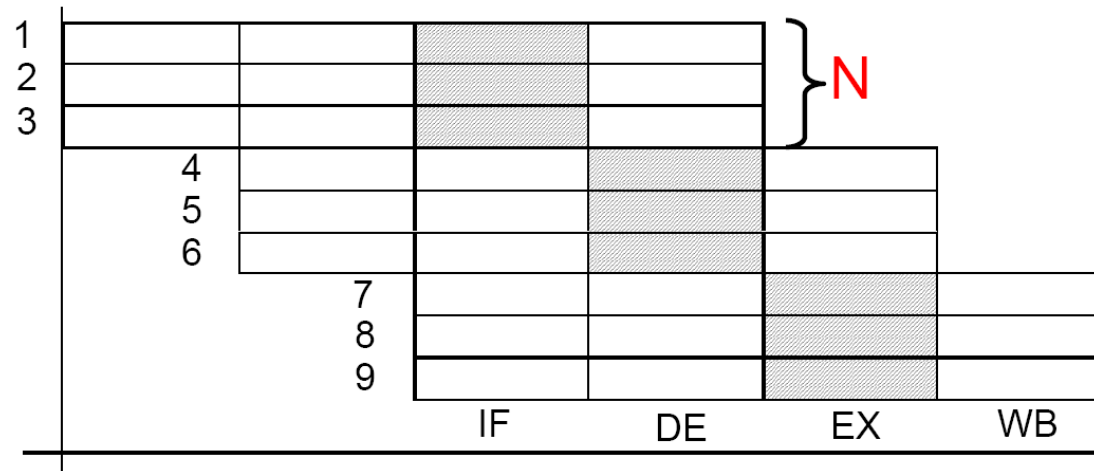
Superscalar Machine

Superscalar (Pipelined) Execution

IP = *DxN*

OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

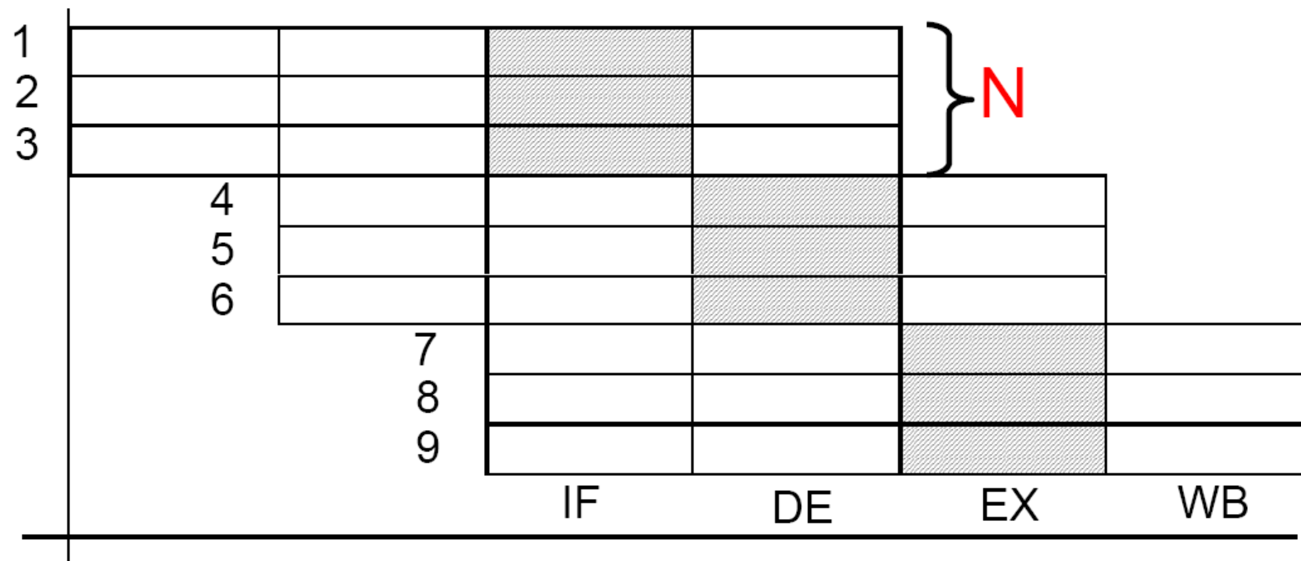
What is the real problem?

CPI of in-order pipelines degrades very sharply if the machine parallelism is increased beyond a certain point.

i.e., when NxM approaches average distance between dependent instructions

Forwarding is no longer effective

Pipeline may never be full due to frequent dependency stalls!



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Missed Speedup in In-Order Pipelines

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`addf f0,f1,f2`	F	D	E+	E+	E+	W										
`mulf f2,f3,f2`		F	D	d*	d*	E*	E*	E*	E*	E*	W					
`subf f0,f1,f4`			F	p*	p*	D	E+	E+	E+	W						

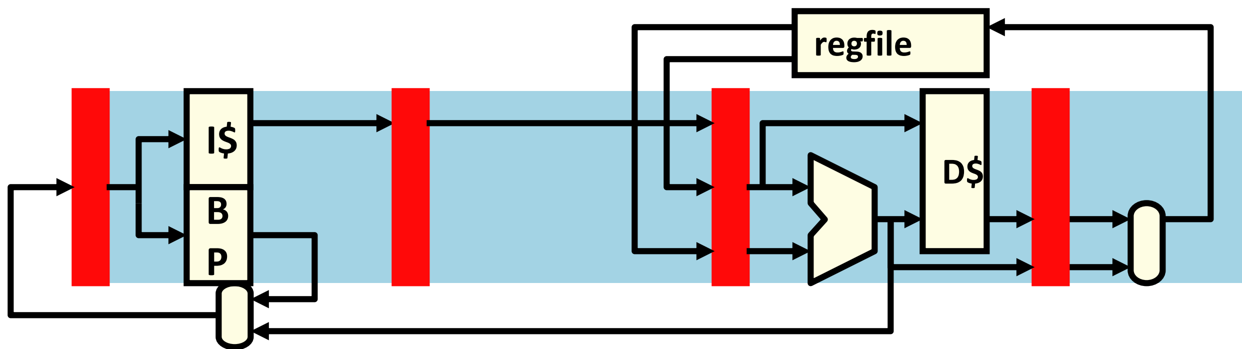
What's happening in cycle 4?

- **`mulf`** stalls due to **RAW hazard**
  - OK, this is a fundamental problem
- **`subf`** stalls due to **pipeline hazard**
  - Why? **`subf`** can't proceed into D because **`mulf`** is there
  - That is the only reason, and it isn't a fundamental one

Why can't **`subf`** go into D in cycle 4 and E+ in cycle 5?

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

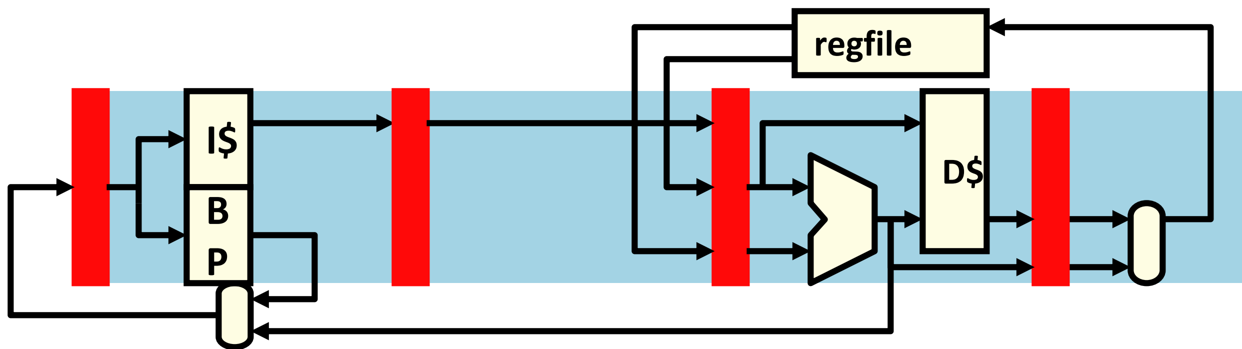
The Problem With In-Order Pipelines



- **In-order pipeline**
  - **Structural hazard**: 1 insn register (latch) per stage
    - 1 instruction per stage per cycle (unless pipeline is replicated)
    - Younger instr. can't "pass" older instr. without "clobbering" it
- **Out-of-order pipeline**
  - Implement "passing" functionality by removing structural hazard

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

New Pipeline Terminology



- **In-order pipeline**
  - Often written as F,D,X,W (multi-cycle X includes M)
  - Variable latency
    - 1-cycle integer (including mem)
    - 3-cycle pipelined FP

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

ILP: Instruction-Level Parallelism

ILP is a measure of the amount of inter-dependencies between instructions

Average ILP =no. instruction / no. cyc required

code1: ILP = 1

i.e. must execute serially

code2: ILP = 3

i.e. can execute at the same time

code1:

```
r1 ← r2 + 1
r3 ← r1 / 17
r4 ← r0 - r3
```

code2:

```
r1 ← r2 + 1
r3 ← r9 / 17
r4 ← r0 - r10
```

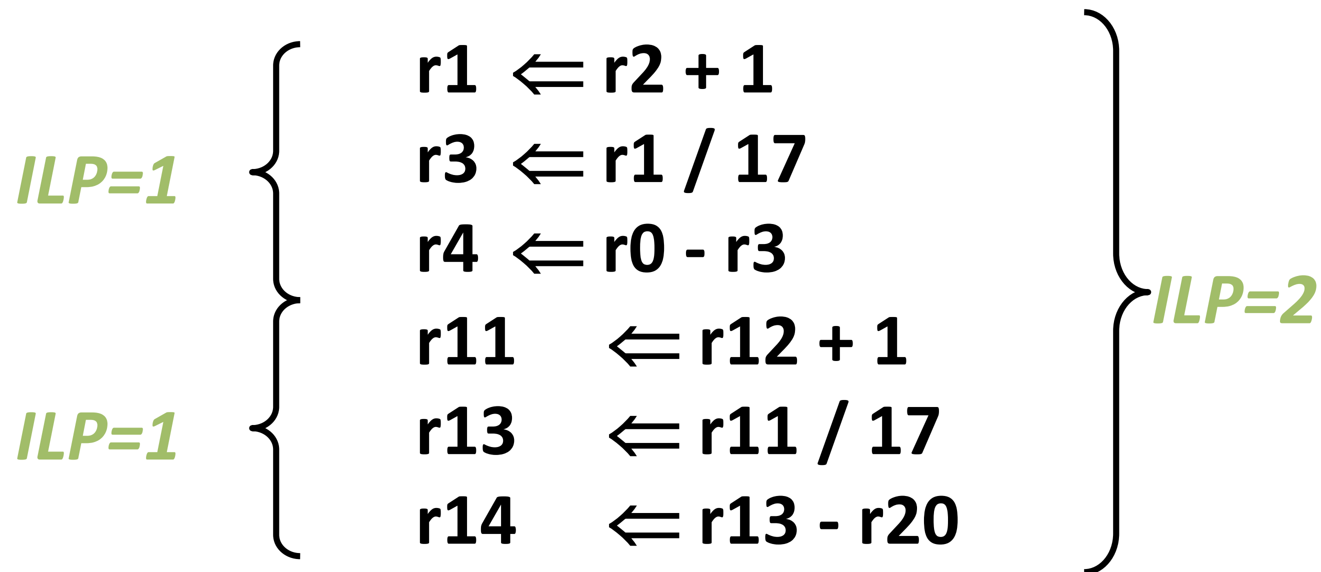
Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Purported Limits on ILP

Study	Limit
Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7
Kuck et al. [1972]	8
Riseman and Foster [1972]	51
Nicolau and Fisher [1984]	90

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

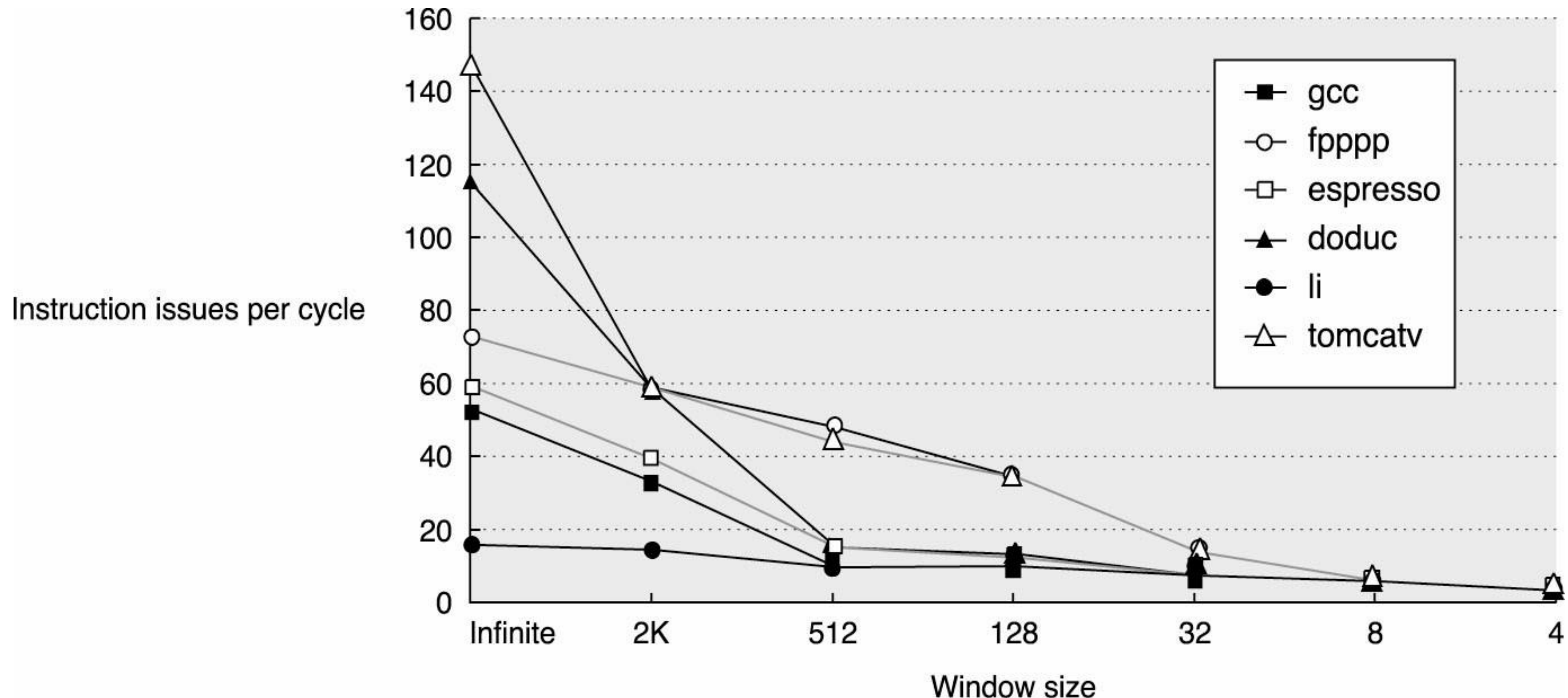
Scope of ILP Analysis



Out-of-order execution exposes more ILP

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

How Large Must the "Window" Be?



© 2003 Elsevier Science (USA). All rights reserved.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Dynamic Scheduling – OoO Execution

- Dynamic scheduling
  - Totally in the hardware
  - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies (WAW and WAR)
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky (much more later)
- Commit instructions in order
  - Any strange happens before commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

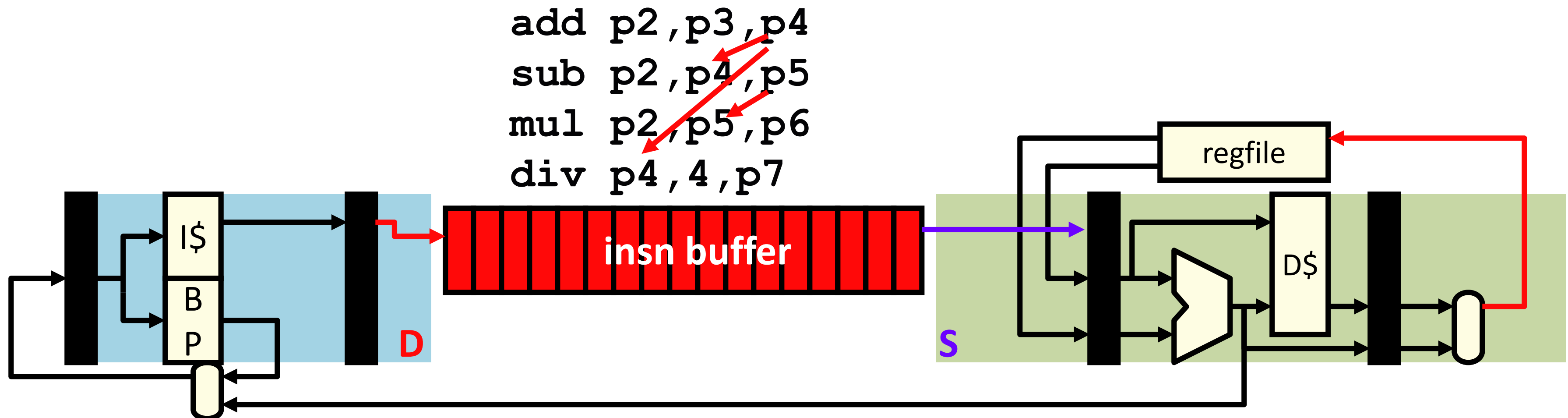
Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Motivation for Dynamic Scheduling

- **Dynamic scheduling (out-of-order execution)**
  - Execute instructions in non-sequential order…
    - \+ Reduce RAW stalls
    - \+ Increase pipeline and functional unit (FU) utilization
      - Original motivation was to increase FP unit utilization
    - \+ Expose more opportunities for parallel issue (ILP)
      - Not in-order → can be in parallel
  - …but make it appear like sequential execution
    - Important
      - But difficult
    - Next few lectures

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Dynamic Scheduling: The Big Picture



Ready Table

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

t

```
add p2,p3,p4
sub p2,p4,p5   and   div p4,4,p7
mul p2,p5,p6
```

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute when ready

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Going Forward: What's Next

- We'll build this up in steps over the next few weeks
  - Register renaming to eliminate "false" dependencies
  - "Tomasulo's algorithm" to implement OoO execution
  - Handling precise state and speculation
  - Handling memory dependencies
- Let's get started!

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Dependency vs. Hazard

- A dependency exists *independent* of the hardware.
  - So if Inst #1's result is needed for Inst #1000 there is a dependency
  - It is only a *hazard* if the hardware has to deal with it.
    - So in our pipelined machine we only worried if there wasn't a "buffer" of two instructions between the dependent instructions.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

True Data dependencies

- True data dependency
  - RAW – Read after Write

R1=R2+R3

R4=R1+R5

- True dependencies prevent reordering
  - (Mostly) unavoidable

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

False Data Dependencies

- False or Name dependencies
  - WAW – Write after Write

R1=R2+R3

R1=R4+R5

  - WAR – Write after Read

R2=R1+R3

R1=R4+R5

- False dependencies prevent reordering
  - Can they be eliminated? (Yes, with renaming!)

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

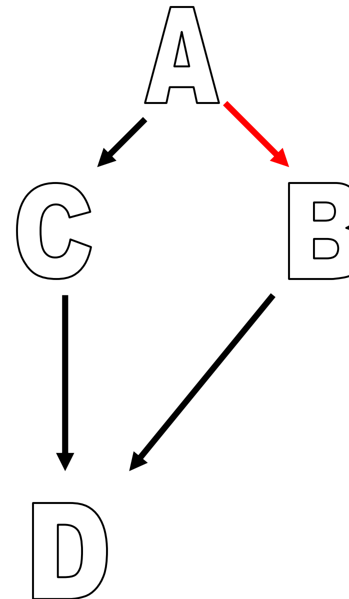
Data Dependency Graph: Simple example

R1=MEM[R2+0] // A

R2=R2+4 // B

R3=R1+R4 // C

MEM[R2+0]=R3 // D

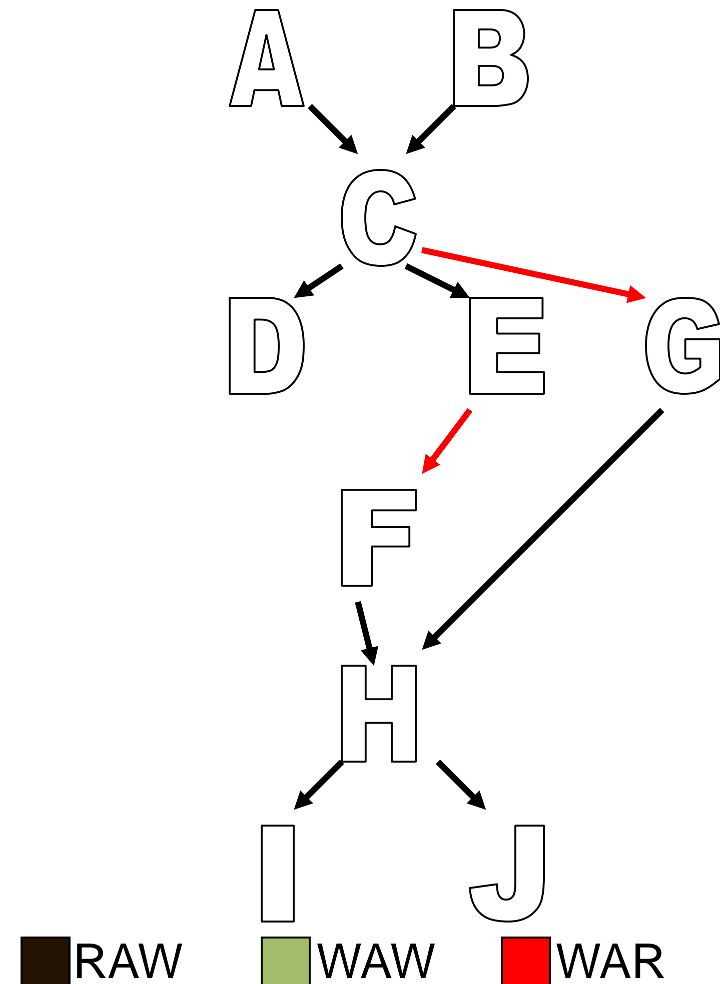


RAW WAW WAR

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Data Dependency Graph: More complex example

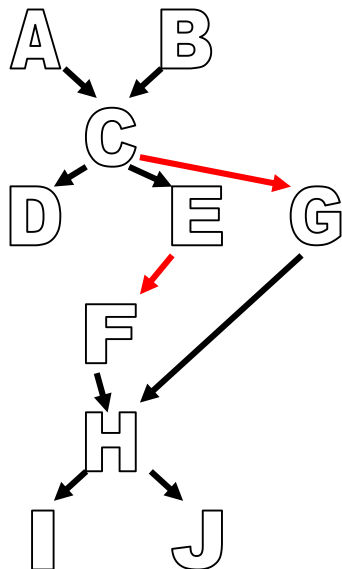
```
R1=MEM[R3+4]      // A
R2=MEM[R3+8]      // B
R1=R1*R2          // C
MEM[R3+4]=R1      // D
MEM[R3+8]=R1      // E
R1=MEM[R3+12]     // F
R2=MEM[R3+16]     // G
R1=R1*R2          // H
MEM[R3+12]=R1     // I
MEM[R3+16]=R1     // J
```



Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Eliminating False Dependencies

```
R1=MEM[R3+4]     // A
R2=MEM[R3+8]     // B
R1=R1*R2         // C
MEM[R3+4]=R1     // D
MEM[R3+8]=R1     // E
R1=MEM[R3+12]    // F
R2=MEM[R3+16]    // G
R1=R1*R2         // H
MEM[R3+12]=R1    // I
MEM[R3+16]=R1    // J
```



- Well, logically there is no reason for F-J to be dependent on A-E. So…..
    - ABFG
    - CH
    - DEIJ
  - Should be possible.
- But that would cause either C or H to have the wrong reg inputs
- How do we fix this?
  - Remember, the dependency is really on the *name* of the register
  - So… change the register names!

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Register Renaming Concept

- The register names are arbitrary
- The register name only needs to be consistent between writes.

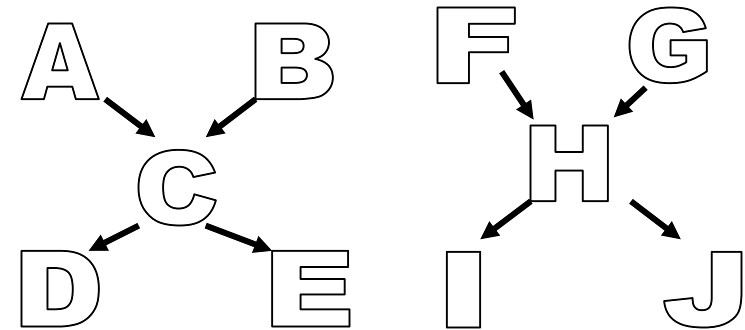
R1= …..
…. = R1 ….
….= … R1
R1= ….

The value in R1 is "alive" from when the value is written until the last read of that value.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

So after renaming, what happens to the dependencies?

```
P1=MEM[R3+4]      //A
P2=MEM[R3+8]      //B
P3=P1*P2          //C
MEM[R3+4]=P3      //D
MEM[R3+8]=P3      //E
P4=MEM[R3+12]     //F
P5=MEM[R3+16]     //G
P6=P4*P5          //H
MEM[R3+12]=P6     //I
MEM[R3+16]=P6     //J
```



RAW WAW WAR

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Register Renaming Approach

- Every time an architected register is written we assign it to a physical register
  - Until the architected register is written again, we continue to translate it to the physical register number
  - Leaves **RAW** dependencies intact
- It is really simple, let's look at an example:
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4**–**p7** are "free"

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

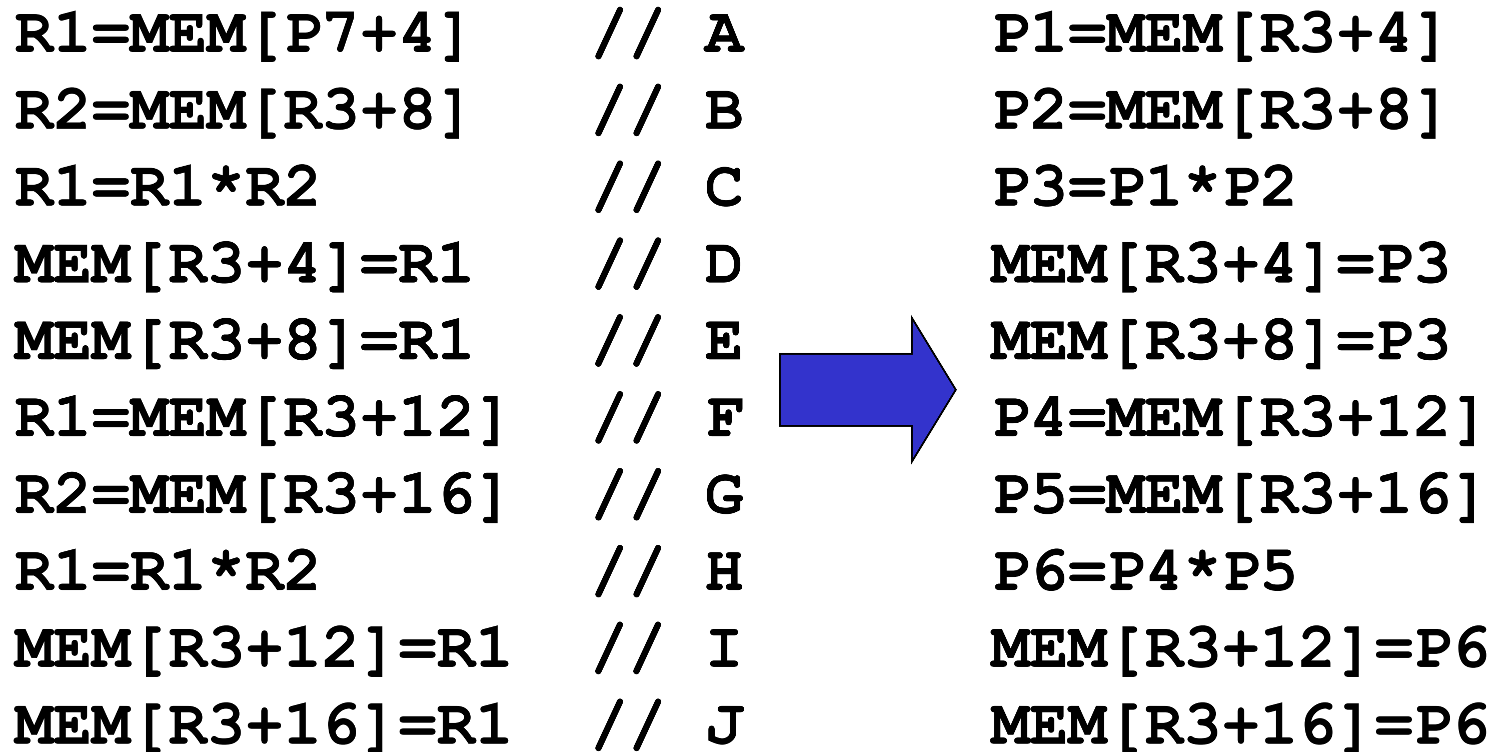
Orig. insns

```
add r2,r3,r1
sub r2,r1,r3
mul r2,r3,r3
div r1,4,r1
```

Renamed insns

```
add p2,p3,p4
sub p2,p4,p5
mul p2,p5,p6
div p4,4,p7
```

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch



Arch	V?	Physical
1	1	
2	1	
3	1	

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Terminology

- There are a lot of terms and ideas in out-of-order processors.
  - And because of lot of the work was done in parallel, there isn't a standard set of names for things.
  - Here we've called the table that maps the architected register to a physical register the "map table". That is probably the most common.
    - I generally use Intel's term "Register Alias Table" or RAT.
    - Also "rename table" isn't an uncommon term for it.
- I try to use a mix of terminology in this class so that you can understand others when they are describing something…
  - It's not as bad as it sounds, but it is annoying at first.

Portions © Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch

Register Renaming Hardware

- Really simple table (rename table)
  - Every time an instruction *which writes a register* is encountered assign it a new physical register number
- But there is some complexity
  - When do you free physical registers?